Microservice integration testing with hardware-in-the-loop in CI/CD pipelines

Master's thesis in Complex Adaptive Systems

Axel Johansson
Simon Paulsson

Department of Mechanics and Maritime Sciences
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2024
www.chalmers.se

Master's thesis 2024

Microservice integration testing with hardware-in-the-loop in CI/CD pipelines

A microservice approach to integration testing in autonomous environments

Axel Johansson, Simon Paulsson

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Mechanics and Maritime Sciences
Division of Vehicle Engineering and Autonomous Systems
Chalmers University of Technology
Gothenburg, Sweden 2024

Microservice integration testing with hardware-in-the-loop in CI/CD pipelines
Axel Johansson, Simon Paulsson

© Axel Johansson, Simon Paulsson, 2024.

Supervisor: Ola Benderius, Department of Mechanics and Maritime Sciences
Examiner: Ola Benderius, Department of Mechanics and Maritime Sciences

Master's Thesis 2024
Department of Mechanics and Maritime Sciences
Division of Vehicle Engineering and Autonomous Systems
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: DoDo testing logo generated by the online service DALL.E 3.

Typeset in LaTeX
Printed by Chalmers digitaltryck
Gothenburg, Sweden 2024

Microservice integration testing with hardware-in-the-loop in CI/CD pipelines
Axel Johansson, Simon Paulsson
Department of Mechanics and Maritime Sciences
Division of Vehicle Engineering and Autonomous Systems
Chalmers University of Technology

Abstract

Mechanical products are increasingly evolving into software-driven systems with expanded responsibilities, as exemplified by the rapid advancements in the vehicle industry and the rise of autonomous driving technologies. Cyber-physical systems (CPSs), such as vehicles that rely on both hardware and software, present significant testing challenges due to their distributed nature and the necessity for real sensor input data. Traditional testing methodologies are suboptimal for CPS, as they typically do not accommodate the integration of both simulation and hardware testing in a distributed environment. This thesis aims to evaluate the feasibility and effectiveness of a containerized, microservice-based testing framework. The framework is designed to support simulation, data replay and hardware test levels and to be integrated into a continuous integration and continuous deployment (CI/CD) pipeline. The proposed framework was implemented and tested within the context of the TME290 Autonomous robots course at Chalmers University of Technology. This involved the development of hardware-in-the-loop (HIL) rigs, as well as the creation of test execution and interpretation software. The framework's capabilities were assessed through the execution of various test scenarios. The integration of the testing framework into the course demonstrated its suitability for simulation, data replay, and hardware test levels on a distributed system. Furthermore, it was successfully integrated into a CI/CD pipeline. The findings suggest that a microservice-based architecture can effectively be used for the integration testing of CPS within a continuous integration environment. This approach enhances the reliability and efficiency of testing processes for autonomous systems, offering a promising solution to the challenges associated with traditional CPS testing methods.

Keywords: Microservices, simulation-in-the-loop, data-in-the-loop, hardware-in-the-loop, end-to-end testing, continuous integration and continuous deployment, GitLab, Docker.

Acknowledgements

We wish to start by expressing our deepest gratitude to our colleague and friend, Norah Andersson Orth, whose efforts cannot be done justice with words. Her contributions include countless iterations in designing and 3D printing parts for the HIL rigs, shopping for essential items, and providing us with tools and hardware, making the workplace more enjoyable, among many other things. Her support, dedication, and patience have been fundamental to the completion of this thesis, which would not be what it is without her. We can only hope that her commitment gave her half as much as it gave us. Thank you, Norah!

Special thanks must be given to our supervisor and examiner, Ola Benderius. Despite being extremely busy, he made time to listen to our ideas and refine them, encouraging us to take on this project. Ola is the reason this project started, and his support, expertise, and financing ensured its completion.

We would also like to acknowledge the efforts of our opponents, Max Sedersten and Amanda Siklund, who have been with us from the beginning, providing us with advice and ideas. Their valuable contributions have helped us improve this work. Finally, thank you to our family, friends and colleagues for your support and office companionship!

Axel Johansson, Simon Paulsson, Gothenburg, June 2024

List of Acronyms

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

CI/CD	Continuous integration and continuous deployment
CPS	Cyber-physical system
DIL	Data-in-the-loop
HIL	Hardware-in-the-loop
SIL	Software-in-the-loop
MSA	Microservice architecture
DevOps	Development and operations
IoU	Intersection over union

Contents

List of Acronyms **ix**

List of Figures **xiii**

List of Tables **xv**

1 Introduction **1**
1.1 Objective . . . 2
1.2 Research questions . . . 2
1.3 Limitations . . . 2
1.4 Outline . . . 2

2 Background **5**
2.1 Problem analysis . . . 5
2.2 Related works . . . 6
2.3 Autonomous robots . . . 7
2.4 DoDo testing . . . 8

3 Theory **9**
3.1 Microservices . . . 9
3.2 Docker . . . 10
3.3 OpenDLV . . . 11
3.4 DevOps . . . 11

4 Method **13**
4.1 Digital environment . . . 15
4.1.1 Software-in-the-loop . . . 15
4.1.2 Data-in-the-loop . . . 16
4.2 Hardware-in-the-loop . . . 17
4.2.1 Static HIL rig . . . 17
4.2.1.1 Hardware description . . . 17
4.2.1.2 Software setup . . . 18
4.2.2 Dynamic HIL rig . . . 20
4.2.2.1 Hardware description . . . 20
4.2.2.2 Software setup . . . 22
4.3 Evaluation layer . . . 24

4.3.1 Perception . . . 24
4.3.2 Open-loop evaluation . . . 26
4.3.3 Closed-loop evaluation . . . 27
4.4 Test registry . . . 28
4.5 Remote integration . . . 29
4.5.1 GitLab . . . 30
4.5.2 DoDo runner . . . 31

5 Results **33**
5.1 Tests . . . 33
5.2 CI/CD integration . . . 33
5.3 Hardware-in-the-loop . . . 36
5.3.1 Static HIL rig . . . 37
5.3.2 Dynamic HIL rig . . . 38

6 Discussion **41**
6.1 Interpretation of results . . . 41
6.2 Implications . . . 42
6.3 Constraints and considerations . . . 42
6.4 Future work . . . 43
6.5 Conclusion . . . 43

Bibliography **45**

List of Figures

2.1 Example structure of a typical cyber-physical system. The system is distributed across three platforms, each performing specific computation steps. Two of the platforms are equipped with sensors, while the third platform generates a response signal to an actuator. 6
2.2 Picture of the Kiwi car developed by Chalmers Revere. This autonomous vehicle, named Norah, is equipped with various sensors and actuators. 7
2.3 Logo for the DoDo testing project. The image was generated by the online service DALL.E 3. 8

3.1 This figure illustrates how the code base is structured in a monolithic- and microservice architecture. The key difference is that logic are separated in small independent units in the microservice architecture. 9
3.2 Illustration over how one node in a network can broadcast a message to multiple nodes, so called *one-to-many* communication. 11
3.3 Illustration of a shared memory between two processes. 12

4.1 The expected division between software parts in a CPS tested with the DoDo framework. 13
4.2 Schematic view over the general test composition. The dashed arrow implicates the optional feedback from the application layer to the action layer. 14
4.3 An example of the view from the simulated camera. 16
4.4 3D-printed holder for the Raspberry Pi 4 and camera module in the static HIL rig. 18
4.5 View from the camera on the static HIL rig. 19
4.6 The finished static HIL rig. 19
4.7 Linear actuator from OpenBuilds. 20
4.8 Final construction of the dynamic HIL rig. 21
4.9 Communication between the Raspberry Pi and the stepper motors. . 22
4.10 Example of a camera feed from the static HIL rig with drawn ground truth bounding boxes. 25
4.11 Geometric interpretation of the IoU metric between two rectangular bounding boxes. 25
4.12 How the target signal $T(t_t)$ is converted into the interpolated target $\hat{T}(t_s)$ aligning with the signal $S(t_s)$ to be evaluated. 27

4.13 Example of the closed-loop evaluation. The red line is from a simulated vehicle that failed the test by leaving the allowed area, and the green line is from a simulated vehicle that reached the finish line without leaving the allowed area. 28
4.14 The frameworks CI/CD pipeline: (1) Push code to GitLab. (2) Build a Docker image. (3) Start end-to-end testing with the DoDo framework. (4) Present results. 29
4.15 Proposed GitLab structure for integration of automated end-to-end testing with the DoDo framework. 30

5.1 Cumulative amount of microservices, M, required to develop T tests. The trend could indicate that as more tests are developed fewer and fewer microservices must be developed per test as the tests can share previous microservices. 35
5.2 Test summary reported in GitLab, summarized by the DoDo runner. 36
5.3 Total test time, split between the setup time and execution time. . . 37
5.4 The final construction and placement of the static HIL rig. 37
5.5 Point of view for the camera. 37
5.6 Static HIL rig camera snapshot for four different environments. 38
5.7 Final construction on the dynamic HIL rig. 39

List of Tables

4.1 Serial commands to the Arduino. 23
4.2 Calibrated parameters for conversion between SI-units and steps. . . 23
4.3 Compatibility of testing methods across different test environments. . 24

5.1 A table with all tests developed during the project. 34
5.2 Description of microservices we used to compose the tests in Table 5.1. 35

1

Introduction

The increase in products incorporating software over the past decade is undeniable, evolving from predominantly mechanical to more digital (cyber) products. These systems, referred to as cyber-physical systems (CPS), integrate computation with physical processes and software components in a deeply intertwined manner [1]. Today, it is rare to encounter a modern system devoid of such technologies; they encompass vital systems like heating, ventilation, air conditioning, water management, and traffic control [2].

As various products, including vehicles, become increasingly software-driven and are entrusted with tasks of greater responsibility, such as autonomous driving, the potential impact of software errors on our safety and well-being escalates [3]. This shift underscores the critical need for meticulous testing procedures, as testing serves as a crucial safeguard against potential errors. However, CPSs introduce new challenges to the testing process.

Testing CPSs presents a significant challenge due to their reliance on both software and hardware integration, rendering complete testing in simulation environments insufficient. Despite this limitation, simulations remain highly valuable, as identifying errors using perturbed data in hardware tests is often impractical. These challenges are further pronounced by the dynamic nature of CPS products, which introduces additional complexities such as code maintainability, scalability, and dependency management. These complexities are particularly pronounced in the context of complex distributed systems like CPSs [2].

In response to these challenges, the microservice architecture (MSA) has gained popularity. This approach organizes an application into a collection of loosely coupled services, which effectively manage the dynamic and complex nature of CPSs [4, 5, 6]. However, while MSA offers notable advantages, it also introduces increased complexity in managing inter-service communication and deploying the application.

Addressing these multifaceted challenges demands a testing strategy or framework that adheres to several key criteria mentioned above. The framework must embody the principles of modularity to efficiently test CPSs while being easily updatable, adjustable, and scalable. Furthermore, to align with the dynamic nature of product development workflows, integration into continuous integration and continuous deployment (CI/CD) is essential. Finally, given the unique characteristics of CPSs, the framework must also be able to accommodate hardware-in-the-loop (HIL) tests.

1.1 Objective

The objective of this thesis is to investigate the feasibility and effectiveness of a containerized, microservice-based testing framework. This framework should support multiple test levels, including software-in-the-loop (SIL), data-in-the-loop (DIL), and, importantly, HIL. Additionally, the framework should be capable of integration into a CI/CD pipeline

Additionally, this thesis will focus on developing specific integration tests for the master's course TME290 Autonomous robots at Chalmers University of Technology. To facilitate these tests, two hardware rigs will be developed: a static rig for evaluating the CPS's environmental perception, and a dynamic rig for assessing the steering algorithm logic.

The primary aim of the course integration is to demonstrate the effectiveness of the framework. By simplifying the testing process, the goal is to help students identify and address bugs more efficiently, allowing them to allocate more time to other course components, thereby enriching their learning experience.

1.2 Research questions

The thesis aims to answer the following research questions:

1. Can a microservice architecture support multi-level testing?
2. Is a microservice architecture suitable for HIL testing?
3. What are the challenges of integrating a microservice architecture in a CI/CD pipeline?

1.3 Limitations

This thesis focused exclusively on investigating the test levels SIL, DIL, and HIL, utilizing Docker for deployment and packaging, and exploring CI/CD integration approaches with GitLab's API. Furthermore, the HIL rigs are designed as a proof of concept, so they have not been fully developed. Finally, the tests developed in this study adhere to the same inter-service communication protocol as in the TME290 Autonomous robots course, based on OpenDLV.

1.4 Outline

The following chapters in this thesis are organized as follows; First, a comprehensive background to the thesis is provided in Chapter 2. This includes an analysis of the problem, a discussion about the course TME290, where we implemented our solution for the students, and a review of related works to give better context to the ongoing research. Next, Chapter 3 covers the necessary theoretical foundation and terminology needed to understand the rest of the thesis. In Chapter 4, our

approach to developing an MSA testing framework is discussed, together with our methodologies for developing multi-level tests and HIL rigs. The results of our work are then presented in Chapter 5, before the report is concluded in Chapter 6, with a discussion about our findings and their implications for future research.

2

Background

In this chapter a brief background to the thesis will be given. The chapter will then outline previous work related to this topic to see where current state-of-the-art testing frameworks built using microservices are today. This will also explain why research around this topic is relevant. Finally this chapter will focus on the background to the TME290 Autonomous robots course integration towards the Kiwi car platform.

2.1 Problem analysis

Software testing is the process in which the quality, functionality, and performance of a software product are assessed before its release to the customer. It is an essential part of software development as it ensures the delivery of safe and reliable software, reduces costs, and improves code quality. Consequently, numerous tools have been developed over the years to help developers automate and expedite this process.

However, most of these tools were originally designed for large and monolithic codebases, rather than the MSA that have become prevalent in recent years. This trend is primarily driven by the increasing use of large, distributed web applications that rely on thousands of microservices spread across multiple computers to deliver their services effectively [7]. Moreover, microservices have also been increasingly adopted in CPSs due to their ability to facilitate the integration of hardware with software systems, addressing the inherent challenges in CPS development [6]. Figure 2.1 illustrates a CPS utilizing a MSA.

Therefore, there is a need for frameworks designed for testing systems based on the MSA [8]. One of the primary challenges lies in the complex testing environment required to evaluate numerous independent microservices [7]. This complexity is further amplified when testing CPSs, as some services depend on hardware components. Consequently, field tests become necessary for adequately testing CPSs as simulations, although beneficial, cannot fully replicate the intricate interactions between CPSs and real-world environments [6]. Together, these challenges make testing of CPSs with a microservice architecture difficult, expensive and time-consuming [8].

Moreover, many companies have adopted agile workflows where code is continuously tested, integrated, and deployed. Hence, such tools must adhere to these principles to ensure functionality and maintain system reliability.

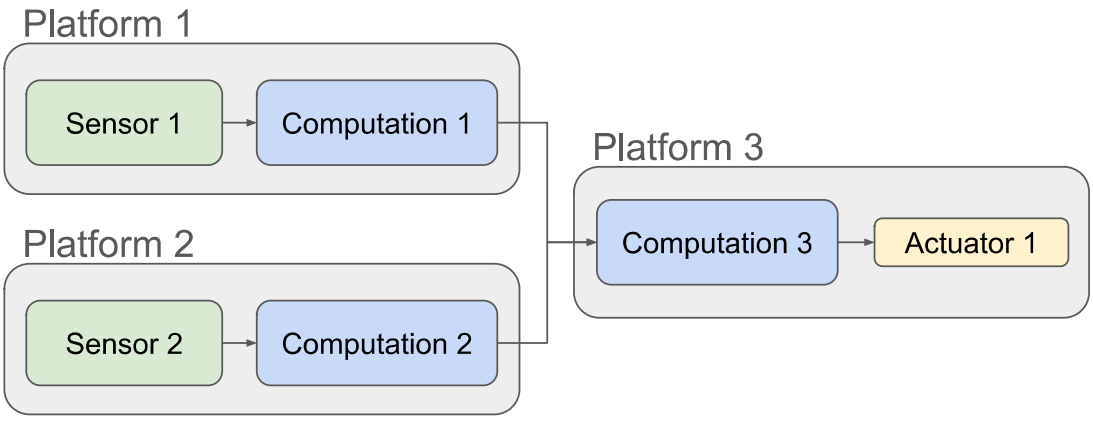

Figure 2.1: Example structure of a typical cyber-physical system. The system is distributed across three platforms, each performing specific computation steps. Two of the platforms are equipped with sensors, while the third platform generates a response signal to an actuator.

2.2 Related works

Applications based on microservices are growing in popularity in the industry. This trend is confirmed through several secondary studies [9, 10]. While the interest in MSA is increasing, some challenges have been identified. As many MSA-based applications can consist of hundreds or sometimes even thousands of individual microservices, these systems can be incredibly complex. This has lead to difficulties in testing and verifying the application, especially in an automated manner [8]. One reason is that the application depends on so many independent parts that it is a challenge to automatically set up a good testing environment.

However, some advantages have also been identified with a microservice-based testing framework. Using a MSA-based application gives unique possibilities for dispatching tests across different testing levels [11]. This possibility comes from the very foundation of microservices as, per definition, they should be independent of each other. Therefore, a microservice-based testing framework can easily replace critical services with a new microservice dedicated to testing during a test. For example, a microservice feeding images from a camera to the rest of the system can be replaced with a simulated camera during testing, without any other microservice noticing the difference.

With this, a microservice-based testing framework can switch out some key services (mainly those dedicated to data collection) during testing. In this way, the main logic of the application can be tested against different scenarios, such as SIL, DIL and HIL, by inserting different microservices based on the test case.

2.3 Autonomous robots

This thesis is integrated into the course TME290 Autonomous robots at Chalmers University of Technology. The course aims to provide students with an understanding of design principles for autonomous systems, along with practical experience through the development software application to a simple autonomous robot, the Kiwi car.

The Kiwi car, developed at the Chalmers laboratory Revere, is intended to promote learning about autonomous vehicles. It is a 3D-printable miniature car equipped with software built on the OpenDLV framework. OpenDLV, detailed in Section 3.3, is a microservice-based software framework for autonomous vehicles, primarily used for sensor and actuator interfacing, as well as inter-service communication [12].

The Kiwi car is equipped with a Raspberry Pi 4, a Raspberry Pi camera module 2, four distance sensors, a DC motor, and a steering servo, see Figure 2.2. The four distance sensors include two infrared sensors placed on the sides of the car and two ultrasonic sensors installed at the front and back of the vehicle [13].

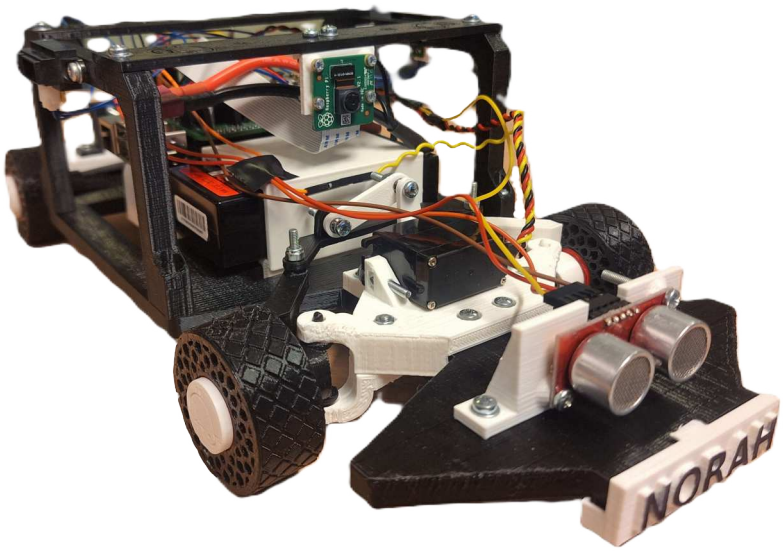

Figure 2.2: Picture of the Kiwi car developed by Chalmers Revere. This autonomous vehicle, named Norah, is equipped with various sensors and actuators.

In the course, students are tasked with developing software capable of autonomous navigation. The primary task is to navigate the Kiwi car around a track made of blue and yellow cones. However, from using the Kiwi car for almost a decade it has been noted that it tends to in some ways, and for some student groups, detract

from some of the learning objectives. Not at least due to hardware problems, lack of laboratory and supervision time.

2.4 DoDo testing

This project is partially initiated in support of the Revere organization's Kiwi platform. Following the naming convention established within Revere, where projects are named after animals [14], this initiative was named *DoDo testing*. This name serves as a double entendre, honoring the extinct and flightless bird, the dodo, while also playfully encouraging active participation in testing activities. This will later be referred to as our framework.

Figure 2.3: Logo for the DoDo testing project. The image was generated by the online service DALL.E 3.

3

Theory

This chapter aims to explore three fundamental pillars of modern software development used in this thesis: microservices, Docker, and development and operations (DevOps). Each section of this chapter delves into one of these key areas, providing insights into their principles and benefits.

3.1 Microservices

The microservice architecture is an architectural style that structures an application as a collection of loosely coupled services in contrast to a monolithic codebase. An illustration of the microservice architecture compared to a monolithic can be seen in Figure 3.1. The microservice architecture is characterized by service independence and that services are based on functionality, where each service is designed to handle a specific task or function, with clearly defined inputs and outputs [15].

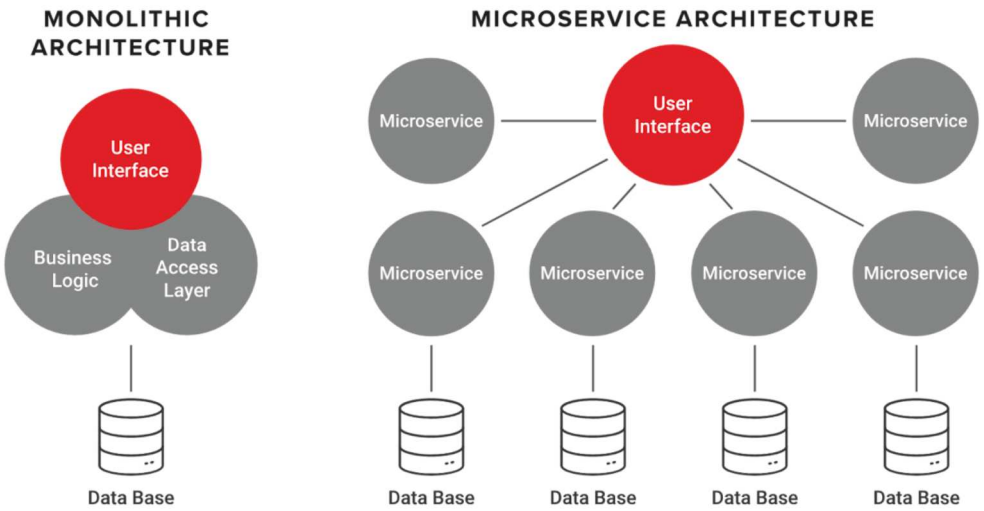

Figure 3.1: This figure illustrates how the code base is structured in a monolithic- and microservice architecture. The key difference is that logic are separated in small independent units in the microservice architecture.

This approach allows an application to be developed, deployed, and scaled as a suite of small, independent services, each running in its own process and communicating with lightweight mechanisms, like user datagram protocol (UDP). This decentralizes

development as it facilitates the distribution of development tasks across different teams, allowing teams to specialize in specific technologies. Having smaller, focused teams to manage individual services enhances agility, leading to faster development cycles and the ability to continuously improve and adapt quickly to changing requirements or market needs.

The independence of services also increases the overall system's resilience to failures and allows for more robust fault isolation and recovery mechanisms. Moreover, it offers flexibility in software deployment, a crucial aspect in adapting to diverse product offerings with varying feature sets. This flexibility enables easy differentiation between budget and premium products, facilitating tailored solutions for different customer segments.

While microservices offer significant benefits, they also introduces complexity and challenges in inter-service communication, data consistency and operational complexity. To combat these challenges tools and best practices like Docker can be helpful [16].

3.2 Docker

Docker, released in 2013, is an open platform for developing, shipping, and running applications in lightweight containers so that applications can work efficiently in different environments in isolation. It is a set of platform as a service (PaaS) products that use OS-level virtualization to deliver software in packages called containers [16, 17]. Docker is a commercial product of Docker Inc. A similar free and fully open-source implementation is Podman, that shares many of the design goals and interfaces as Docker.

A container is an isolated environment, and bundle their own software, libraries and configuration files. They have everything that the code needs in order to run, down to a base operating system. This is one of the greatest benefits as containers can be run on any operating system with any processor, simplifying the deployment process by manifolds. The isolated environment causes the containers to have no knowledge of your operating system, or your files or other containers by default. Because all of the containers share the services of a single operating system kernel, they use fewer resources than virtual machines [18].

The Docker platform is highly relevant when working with microservices as a large challenge there is the deployment process. Docker simplifies this process via the `docker compose up` command. This command is typically used on a compose file. This is a YAML file, usually named `docker-compose.yml`. It holds the configuration of the application as it specifies the images and volumes that will be used [19].

The images are an executable package that determines how to create a container, what software it will use, and how it will run, and the volumes specifies what data to share between container and the host file system. From these the the multi-container Docker applications is defined, configured and executed.

Thus Docker helps to streamline the deployment and testing process of microservices by allowing developers to define complex, multi-container setups in simple, declarative YAML files, cross-compiled and deployable across different platforms and environments.

3.3 OpenDLV

OpenDLV was initiated in 2015 at Chalmers University of Technology, as a software intended for autonomous vehicles at the Revere laboratory [12]. OpenDLV is built upon the Libcluon messaging library by Prof. Christian Berger. [20]. The library is designed to quickly set up a distributed software system where microservices can exchange messages. The library supports multiple modes of communication, including UDP, TCP and shared memory.

The DoDo-framework is built on the Libcluon library, utilizing the UDP and shared memory communication. The UDP session allows for *one-to-many* communication, which is illustrated in Figure 3.2. In this type of communication, one node can send messages to multiple other nodes in the network.

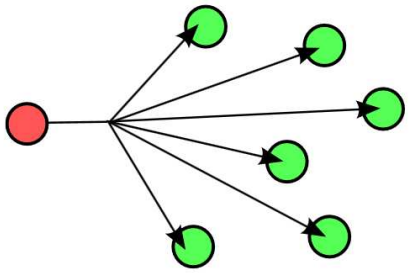

Figure 3.2: Illustration over how one node in a network can broadcast a message to multiple nodes, so called *one-to-many* communication.

This capability proves beneficial within a distributed system reliant on microservices, enabling them to exchange messages without requiring knowledge of which other services are listening for the messages. This means that the microservices are completely decoupled and can easily be replaced or removed from the system without anything breaking.

Shared memory is also a method for sharing data between processes, limited to the same device. Illustrated in Figure 3.3, this method enables two processes to access the same physical memory, promoting rapid communication without intermediary steps.

3.4 DevOps

DevOps is a methodology that fosters collaboration, automation, and integration between development and operations teams to streamline software delivery. Continuous integration (CI) involves frequently integrating code changes with automated

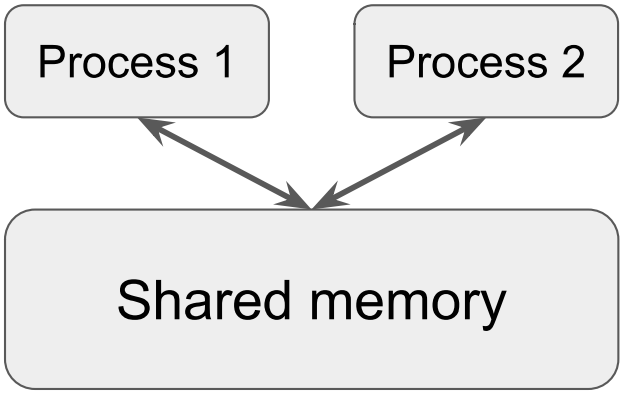

Figure 3.3: Illustration of a shared memory between two processes.

builds and tests to detect problems early. Continuous delivery (CD) automates deployment for faster release cycles. DevOps promotes a microservice architecture for agility and scalability, enabling faster, more efficient, and reliable software delivery.

Distributed version control, such as Git, is a is a crucial tool within DevOps, serving as the backbone for code version management, enabling seamless collaboration, automation, and continuous integration and delivery (CI/CD). It allows teams to track changes, manage code versions, and automate deployments, which is essential to streamline development and operations processes.

GitLab extends Git's capabilities with a web-based Git repository manager that offers built-in CI/CD pipelines, issue tracking, code review, and collaboration tools. This integration streamlines development workflows, enhancing code management, automated testing and deployment, issue tracking, and team collaboration.

The GitLab runner automates builds in GitLab's CI/CD pipeline, executing jobs from the `.gitlab-ci.yml` file, handling tasks like compiling code, running tests, and generating artifacts. It ensures consistent and efficient builds, integrating seamlessly with GitLab's features. The GitLab runner can be deployed on any machine, connecting to the GitLab server to continuously receive and execute jobs, accommodating complex tasks requiring dedicated hardware.

4

Method

This chapter will in detail describe the steps that were followed to achieve the objective of this thesis. To recap, the objective of the thesis was to construct an end-to-end testing framework, DoDo testing, using a microservice architecture that could support multiple test levels. The test levels studied were SIL, DIL and HIL, with some extra focus placed on the integration of HIL testing with the framework. Additional efforts were made to also integrate the framework with a CI/CD pipeline to automate testing.

In the end the framework was to be demonstrated for the students in the course TME290 Autonomous robots at Chalmers University of Technology, with the Kiwi car used as the CPS platform. This goal had a large impact and can clearly been seen in several design choices throughout the project. Three of the bigger decisions this has influenced were: (1) the use of Docker to containerize and distribute microservices, (2) the use of OpenDLV to communicate between them, and (3) the use of GitLab as a back-end when integrating the framework with a CI/CD pipeline.

All three of these decisions were made primarily to easily integrate the framework with the Kiwi car and the course, as the vehicle already used both Docker and OpenDLV, and the students used GitLab in the course. However, all three decisions have proved advantageous as will be seen in the following sections. Before going into detail about how we developed the framework however, a theoretical view of how the DoDo framework expects a system under testing to work is required.

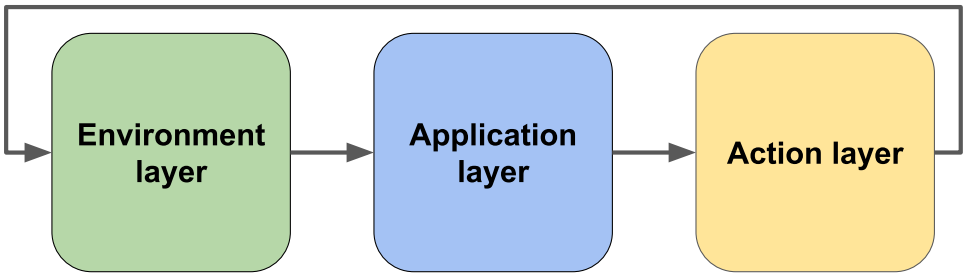

Figure 4.1: The expected division between software parts in a CPS tested with the DoDo framework.

The DoDo framework is meant to test CPSs, which means systems are expected to have a way to interact with their environment. At a minimum a system is expected

to have three microservices, separated into three different layers, environment, application and action, as in Figure 4.1. The environment layer is software that is directly controlling hardware used to gather information about the environment such as cameras or other sensors. The application layer is were the main logic of the CPS happens, and actions are decided based on the perceived environment. Finally the actions are performed by the action layer by interacting with hardware components in CPS. This will generate a change in the environment which can be viewed as a feedback loop to the environment layer.

The reason for this distinction between layers become clear when viewing the DoDo frameworks test structure. The framework will always target the application layer when testing and the environment and action layer will be partially or fully replaced with other microservices to create a controlled test environment. Furthermore, an additional layer that is used to evaluate the test is added as in Figure 4.2. The dashed arrow indicates an optional connection between the application and action layer, depending on if the test is open (no feedback) or closed (with feedback).

The fundamental idea behind this test structure is that the application layer can not distinguish between a test and real environment. All layers are also easily replaceable, making it possible to seamlessly switch between test levels by adding/removing additional microservices to/from the different layers.

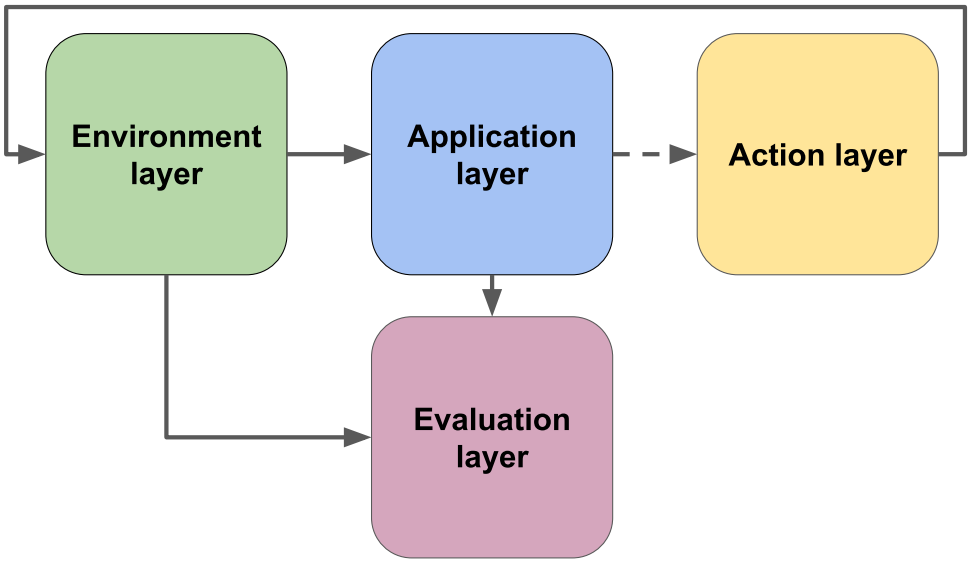

Figure 4.2: Schematic view over the general test composition. The dashed arrow implicates the optional feedback from the application layer to the action layer.

This is the fundamental idea behind the DoDo testing framework and to build it the method followed these six distinct steps:

1. **Digital environment:** Development of environment microservices for simulated (SIL) and real but replayed (DIL) environments.
2. **Hardware rigs:** Creation of an environment built up from real hardware to enable HIL testing. This required planning and building of the hardware rigs.

3. **Evaluation services:** Development of the evaluation layers microservices.
4. **Test deployment:** Formalisation and deployment of the tests. Here the environment and evaluation services was brought together to form the test and deployed in the `DoDo test registry` [21].
5. **CI/CD integration:** Integration of system in CI/CD pipeline for remote and automatic testing. This involved creating the `DoDo test runner` [22] software, acquiring a server, and configuring GitLab.

In the following sections, we expand in order on each of the steps and explain the corresponding components of the system, interactions, data flow, key features and provide usage examples.

4.1 Digital environment

The development of the digital environment, encompassing both SIL and DIL, starts with the condensation of functionality and requirements down to a microservice level. As previously stated all microservices were packaged with Docker and communicates between each other through the OpenDLV protocol. The subsequent sections will provide an overview of the specific requirements for SIL and DIL environments, detailing the necessary components and functionalities. Discussion on hardware-in-the-loop (HIL) testing will be reserved for its own section 4.2.

4.1.1 Software-in-the-loop

SIL testing provides a simulated environment where the real hardware in a CPS is replaced with software that replicates the functionality of the hardware components. In the context of an ideal CPS, as in Figure 4.1, this means that the services in the environment and action layer are fully replaced with microservices simulating the same functionality.

The SIL tests purpose is to pinpoint algorithmic errors, leveraging ideal and controlled sensory data. They provide key benefits linked to test scalability and speed as SIL tests can be run on multiple computers, not restricted to real time. They can also guarantee consistency in the simulated environment over time and between tests as the simulation can be made completely deterministic. Furthermore, SIL testing helps identify and fix bugs and optimize software performance before integration with the actual hardware, reducing development time and costs.

The simulated environment should provide the application layer with the same sensory inputs as in the real environment. In the course the students input to their application is based on the camera feed from the camera module on the Kiwi car. Therefore an environment service that generates camera frames and saves them to shared memory, as the real Kiwi car does, was essential in all SIL tests.

The microservice used to simulate the camera feed is called `virtual-camera` [23]. It builds up the environment by mapping object files (encoding 3D objects and 2D

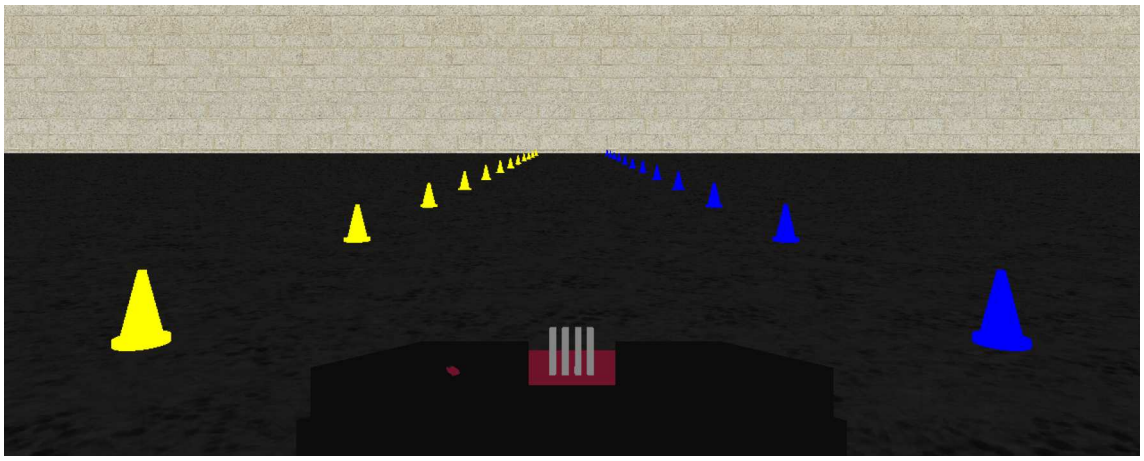

Figure 4.3: An example of the view from the simulated camera.

textures) and a configuration file with object placements. This makes SIL environment very flexible as the environment is generated from a single configuration file, providing an simple way to generate multiple test environments. In Figure 4.3 the point of view is illustrated from the simulated camera.

Another essential functionality is the movement of the vehicle. The approach to achieve this differs based on if the simulation is open- or closed-loop. In the open-loop scenario, no feedback is given to the environment, meaning that the environment layer in the test environment is independent from the application layer being tested. This provides a test environment that will always look the same to the application layer, independently of what the application layer does, providing a stable test environment. The movement is controlled by a single microservice, called `beacon` [24], continuously sending out OpenDLV messages with the current pre-configured position of the vehicle at a fixed frequency.

In the closed-loop scenario however, the feedback loop to the action layer in Figure 4.2 is closed. The `beacon` [24] service is then replaced with a dynamic model of the Kiwi car, `dynamic-model` [25] and an `integrator` [26] service that together provides realistic movement in the test environment based on the application layers output.

4.1.2 Data-in-the-loop

A DIL test, in contrast to a SIL test does not use simulated data. Instead it replays pre-recorded data back to the system being tested. The pre-recorded data is recorded on the CPS when it is being controlled manually or autonomously by another algorithm. All communication between microservices during the run is captured into a `rec` file. Afterwards the `rec` file can then be replayed for another system and the response can be investigated to evaluate the performance of the new system.

To run a DIL test, all microservices in the environment and action layers therefore are replaced with a single microservice that replays a `rec` file for the application layer. This means that in contrast to a SIL test the application layer is exposed to

real world data, something that can help adjust parameters in the application layer to real world conditions, enhancing robustness in the field.

4.2 Hardware-in-the-loop

The HIL testing is meant to complement the SIL and DIL testing that was explained in the previous section, and serve as a natural progression as the application being tested can be exposed to a more realistic environment. The difference from a field test is that the environment should be controlled and reproducible. The HIL rigs enable validation, optimization and verification on a physical system without expensive field testing reducing development time and costs.

The implementation of HIL testing was done in two parts, first with a static HIL rig to demonstrate the functionality of our method, and secondly with a dynamic HIL rig for more advanced testing. The setup of each HIL rig consisted of three steps. The first step was to plan the test, followed by the design and construction of the HIL rig. Finally, the software was built and the HIL rig was installed with all the necessary files and applications. The following sections will provide details about each of these steps for both or static and dynamic HIL rig.

4.2.1 Static HIL rig

The stationary HIL rig was designed to serve as a testing platform to help evaluate the performance and accuracy of the perception part of the application layer. Operating within a controlled environment and utilizing real-world sensors, the rig enables the validation and optimization of the perception algorithm's functionality without the necessity for extensive field testing.

The HIL rig fulfills several key objectives: Firstly, it provides a means for validation of the detection algorithm. The use of real sensors in a static environment ensures consistent output. This enables iterative refinement of the algorithm and the fine-tuning of parameters to enhance its detection accuracy and robustness across software versions.

Furthermore, the rig's lightweight and mobile design enable easy replication with different configurations or relocation to alternative environments. Consequently, it offers a controlled setting for testing the camera detection algorithm under diverse conditions, including variations in lighting and background clutter.

Lastly, the rig serves as a valuable tool for data collection. Its mobility, combined with static object placement, enables the generation of large datasets in various environments with labels. These datasets can then later be used to train, refine or validate detection algorithms.

4.2.1.1 Hardware description

The design of the static HIL rig was centered around a Raspberry Pi 4, connected with a camera module v2, to replicate the equipment found in the Kiwi cars used

in the course. The Raspberry Pi and camera module are securely mounted on a custom-designed 3D-printed holder, see Figure 4.4.

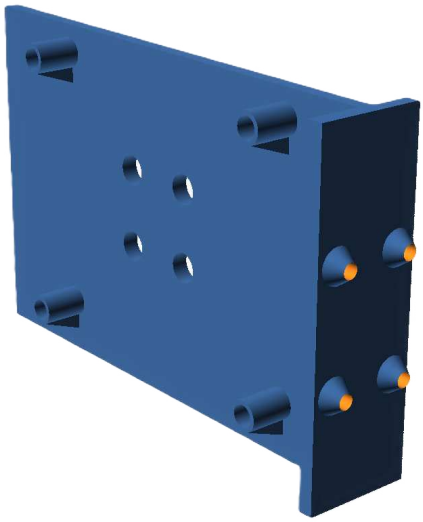

Figure 4.4: 3D-printed holder for the Raspberry Pi 4 and camera module in the static HIL rig.

The 3D-printed holder was screwed onto a wooden block to firmly attach it in a static position to assure a static camera feed from a height resembling the height the camera is mounted on in the Kiwi cars. The wooden block was screwed into a plywood base with the measurements 60×70 cm which is also the HIL rigs footprint, making it relatively small and manageable. From there the camera had most of the plywood base in view and eight 3D-printed cones, four blue and four yellow, were glued onto the base in the cameras view.

The cones were of the same size and color as the ones used in the course TME290, and were placed strategically to find common bugs in the student code. The two difficulties that were introduces were a cone on the edge of the image, and overlapping cones. This can be seen in Figure 4.5, a picture taken with the camera module on the HIL rig.

The Raspberry Pi in the HIL rig is powered with a standard Raspberry Pi power supply and connected to the internet trough an ethernet cable. The final setup can be seen in Figure 4.6.

4.2.1.2 Software setup

With a finished static HIL rig next step was to develop the microservices for the test. For the static HIL test the setup is easy as very few microservices are necessary. The environment layer consists of a single microservice, `opendlv-device-camera-rpicamv2`, feeding frames from the camera module into shared memory for the application layer to handle.

Figure 4.5: View from the camera on the static HIL rig.

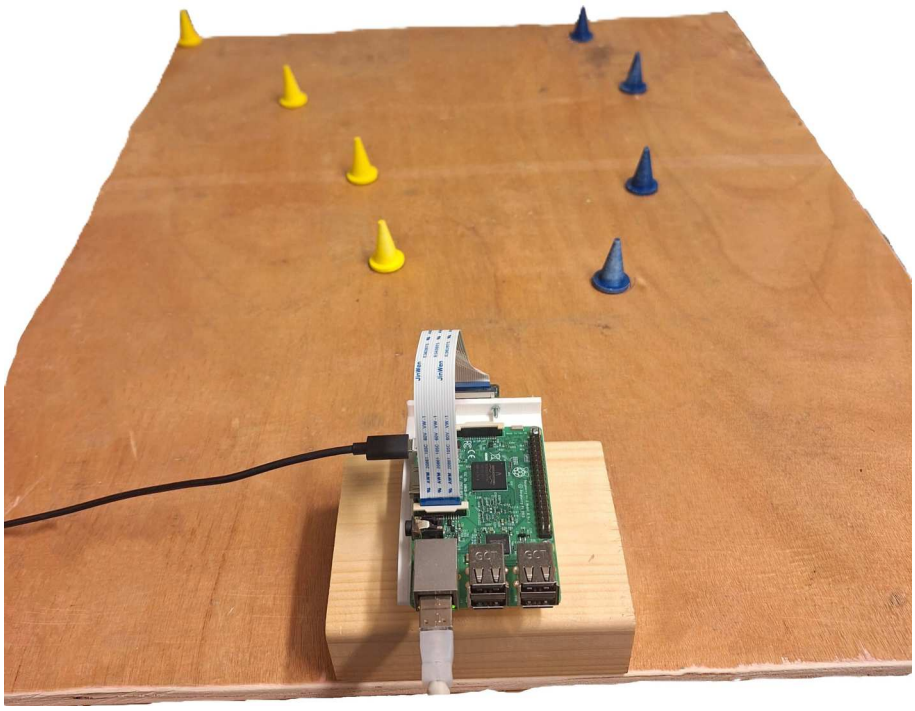

Figure 4.6: The finished static HIL rig.

4.2.2 Dynamic HIL rig

They key purpose of the dynamic HIL rig is to enable automated end-to-end testing, focused on the logic part of a CPSs application layer. A camera with three degrees of freedom (two transnational and one rotational) enables closed-loop testing with real sensory feedback. This means that the logical algorithms controlling the Kiwi cars steering can be tested in a realistic environment.

This helps testing the accuracy and robustness of the steering algorithms, in a repeatable and controlled environment, further reducing the need for early field testing. Just as the static HIL rig, the dynamic HIL rig can also be used for the generation of large and labeled dataset. The datasets can then be used to train, refine and validate steering algorithms for a Kiwi car. The rig can even be used for live-training of algorithms that rely on iterative evaluation with different parameters, such as the *genetic algorithm*.

4.2.2.1 Hardware description

The dynamic HIL rig can be separated into three distinct parts, the base, center-hub and the electronics. The base is the main frame, responsible for the movement of the center-hub. The center-hub is the brain of the dynamic HIL rig and also where the camera is mounted. The electronics connects these two parts together and delivers power to all the electronics.

The base is assembled from three linear actuators from OpenBuilds, see Figure 4.7. The linear actuators are assembled in an **H**-formation, see Figure 4.8, to enable two dimensional movement. The sides are 1500 mm and the middle is 2000 mm. Each actuator is equipped with a NEMA 17 hybrid bipolar stepping motor with an 1.8° step angle (200 steps/revolution). Each phase of the stepper motors draws 1.7 A at 2.8 V, allowing for a holding torque of 0.36 N m. A fourth stepper motor of the same type is mounted on the central actuator. This stepper motor is pointing down and on the shaft the central-hub will be mounted to enable rotational freedom for the camera.

Figure 4.7: Linear actuator from OpenBuilds.

To protect the stepper motors from going outside of the allowed range of operation two limit switches were added for each motor. For the three linear actuates, the limit switches were mounted on the end of each V-SLOT, restricting the allowed transnational movement to a space of $140 \times 190\,\text{cm}$. For the fourth stepper motor, controlling the rotational freedom, the limit switches were placed on top of each other, in opposite directions, restricting the rotational freedom from $-170°$ to $170°$, with $0°$ being forward.

Both the limit switches and the stepper motors require multiple cables to function correctly, and with the moving parts cable management is important to avoid accidents such as cables getting stuck and breaking the dynamic HIL rig. Therefore two cable chains that can move with the motors were mounted on the sides of two of the actuators. All cables going to moving parts were then dragged through these chains before going to the final location.

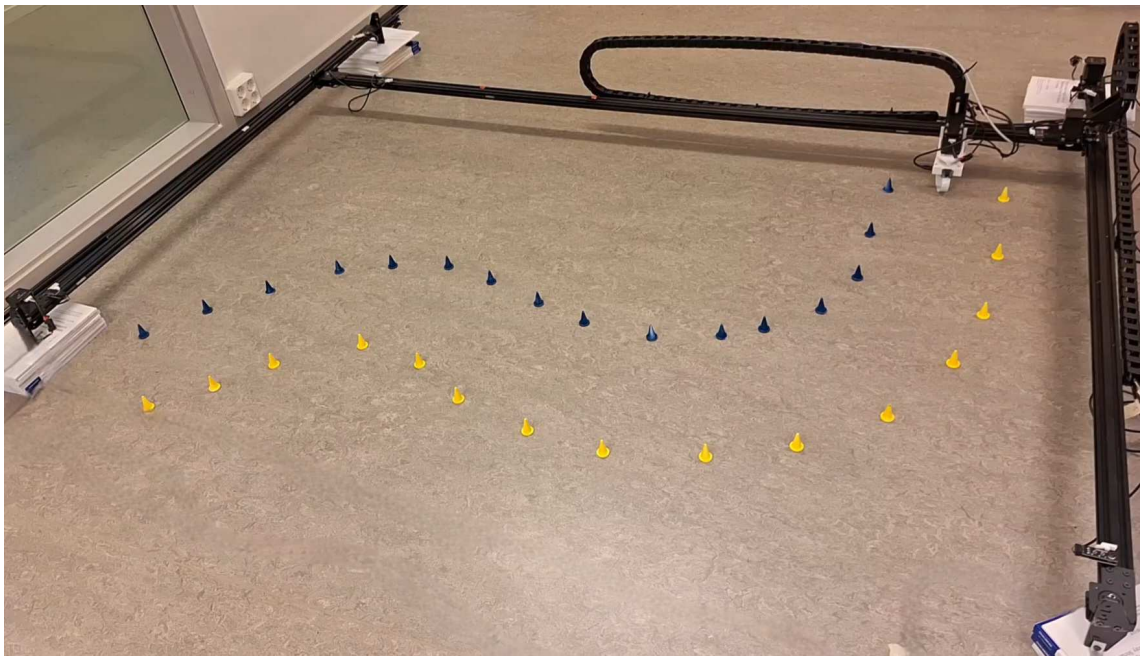

Figure 4.8: Final construction of the dynamic HIL rig.

The center-hub is constructed in the exact same way as the static HIL rig, with the same parts and holder, see Figure 4.4. Instead of mounting the holder to a wooden block though, the holder was mounted onto a metallic plate that could be fastened on the rotor-shaft of the downwards-pointing stepper motor on the base.

The electronics controlling everything consists of several parts. The main parts are an Arduino UNO R3 and four stepper motor controllers (L6470 AutoDriver from SparkFun). The Arduino is connected to the Raspberry Pi on the center-hub through a USB cable, and is used as an interface between the Raspberry Pi and the stepper controllers. Motion commands for the stepper motors are sent through serial communication from the Raspberry Pi to the Arduino, were the motion is converted from natural SI-units to internal stepper motor units (using steps). The converted motion command is then passed out to the stepper controllers through

SPI communication. Each stepper controller finally sends out pulses to each stepper motor performing the requested motion. An overview of the communication can be found in Figure 4.9. The figure also shows how the limit switches are connected directly to the stepper controller so that it can perform a hard stop anytime they are triggered.

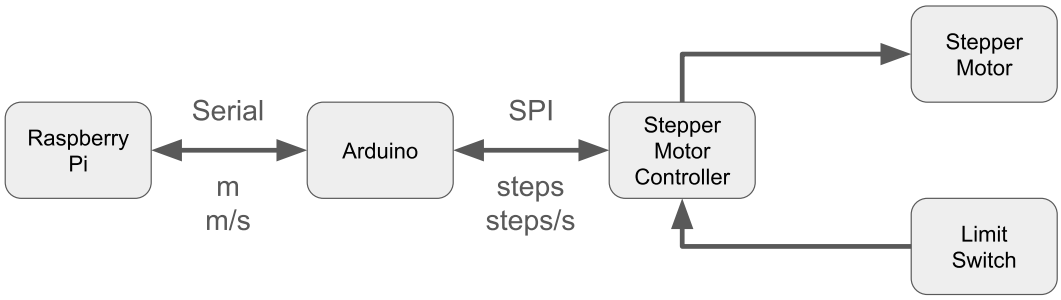

Figure 4.9: Communication between the Raspberry Pi and the stepper motors.

4.2.2.2 Software setup

When the dynamic HIL rig was constructed it required some software before it could move. The software consists of two parts, the Arduino software, responsible for the low level manipulation of the stepper motors, and the Raspberry Pi software, responsible for the high-level movement and dynamics of the simulated Kiwi car.

The Arduino software was developed first and was written to receive commands through serial communication. The commands were written in a human readable format so that the Arduino software could be interfaced directly from a serial interface, without the Raspberry Pi software. This made debugging of the Arduino software much easier.

All commands followed the structure of `$COMMAND:ARG1,ARG2,...\n`. In other words all commands started with an `$` character to denote the beginning of a new command. This was followed by the name of the command and an `:` character to separate the command from the arguments. Each argument was separated with a `,` character and all communication ended with a new-line character. Any command that requests information from the Arduino receives a response with the following format, `$ARG1,ARG2,...\n`. All arguments and responses were given in SI-units (e.g. m, rad and s). In Table 4.1 all commands and their intended purpose are listed.

On the Arduino the motion commands were converted from meters and radians to steps and sent out through SPI to the stepper motor controllers. The conversation from a position in meters (or radians) to a position in steps was made with the function f and the conversion from a velocity in meters (or radians) per second to steps per second through the function g, both defined below.

$$y_i = f(x_i) = x_i \times c_i + \Delta_i, \tag{4.1}$$

$$w_i = g(v_i) = v_i \times c_i, \tag{4.2}$$

Table 4.1: Serial commands to the Arduino.

Command	Description
GOTO	Move the dynamic HIL rig to the position given by the three arguments (x, y and yaw).
RUN	Move the dynamic HIL rig with the given velocities for each degree of freedom (x, y and yaw).
STOP	Stop all movement. No arguments.
CALIBRATE	Calibrate and reset the dynamic HIL rig. No arguments.
POSITION	Get the current position from the rig. No arguments. Receives the position for each degree of freedom (x, y and yaw).
STATUS	Get the status of the dynamic HIL rig. No arguments. Receives an integer corresponding to the current status.

with c_i a conversion parameter unique for each stepper motor that converts from meters (or radians) to steps. The index i can take one of four possible values from x, y_1, y_2 and α, representing the corresponding motor. c_i is a calibrated parameter that is measured in full steps per meter (or radians for $i = \alpha$). Δ_i is an offset measured in steps centering the position $x = y = \alpha = 0$ in the middle of the dynamic HIL rig with the camera pointing forward. In Table 4.2 all calibrated parameter values can be found.

Table 4.2: Calibrated parameters for conversion between SI-units and steps.

i	x	y_1	y_2	α
c_i	3320	3320	3320	0.55
Δ_i	2988	2324	2324	83

When the Arduino software was working the software for the Raspberry Pi was developed. The modular approach of the framework made this development easy as only a single new microservice to control the dynamic HIL rig had do be built. Otherwise both the environment layer and evaluation layer was copied from a closed-loop SIL test, with the exception of the `virtual-camera` [23] microservice that was replaced by the real camera microservice, `rpicam` [25], and the integrator microservice, `integrator` [26], as the integration was done directly by moving the dynamic HIL rig with a given speed. This decision was made so that the simulated and real position would not diverge, if the dynamic HIL rig could not keep up with the simulated speed. Instead the vehicle position was read directly from the dynamic HIL rig.

The newly developed microservice, `dynamic-hil-controller` [27], therefore listened for OpenDLV velocity messages that it forwarded to the Arduino and the stepper motors. It also read the current stepper motor positions from the stepper controllers, via the Arduino and sent them out as OpenDLV position messages.

4.3 Evaluation layer

With three different environments (DIL, SIL and HIL) developed, the next step was to develop a method to automatically assess the performance of the application layer in the test environment. This is were the last layer in Figure 4.2, the evaluation layer, comes into play. The purpose of the evaluation layer is to asses the performance of the CPS and offer valuable feedback to the developer. It monitors both the environment layer and the output from the application layer through the OpenDLV communication protocol, and tries to evaluate the application layers performance.

A common pattern in CPS is to divide the application layer into two parts, here named perception and logic. Each part can consist of multiple microservices that together aims to achieve a common goal. The perception part is meant to take the output from the environment layer and convert it into a form that can be understood and processed by the logic part of the system. The logic part is then responsible for deciding on the appropriate action given the pre-processed data from the perception part. As the course TME290 favours this type of partitioning of the application layer we decided to develop tests that tested these two parts.

A perception test evaluates the effectiveness of the application layer in identifying objects within the test environment, while a logic test assesses how well the application layer responds to various scenarios. Logic tests can be further divided into two distinct subcategories: open-loop and closed-loop, depending on whether the test environment has a closed or open feedback loop. The compatibility between different test environments and test categories is summarized in Table 4.3. Note that only the DIL test is incompatible with closed-loop testing.

Table 4.3: Compatibility of testing methods across different test environments.

	Perception	Open	Closed
SIL	Yes	Yes	Yes
DIL	Yes	Yes	**No**
HIL	Yes	Yes	Yes

The subsequent sections will provide detailed explanations of the three evaluation groups, going in to detail about requirements, use cases and evaluation methods.

4.3.1 Perception

The objective of the perception test, as previously stated, is to evaluate how well the application layer can identify objects in the test environment. In the context of this thesis and the Kiwi car this means how well the application layer can identify objects from a camera frame taken with the camera module on the Kiwi car.

The evaluation service is called `object-detection` [24] and more specifically it compares the application layers output of cone placement and type with the ground truth, see Figure 4.10. The application service communicates this via the OpenDLV messages: `DetectionBoundingBox` and `DetectionType`, which serves as input to the evaluation, together with a configuration file of the ground truth.

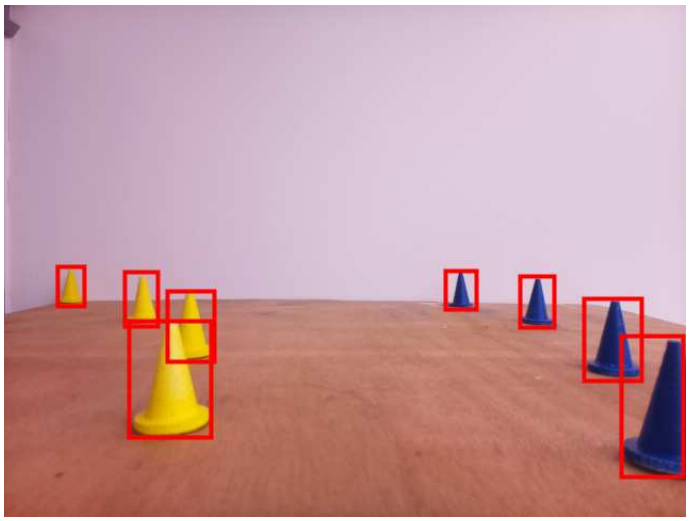

Figure 4.10: Example of a camera feed from the static HIL rig with drawn ground truth bounding boxes.

The bounding box follows the standard convention used in image processing, the Cartesian pixel coordinate system with an inverted y-axis. In this convention, the origin (0,0) is located in the upper-left corner of the image and (1, 1) is the bottom-right corner.

The comparison between bounding boxes was done with the *intersection over union* (IoU) metric, a common metric used in image classification [28]. The metric is a ratio between the intersection area between the two bounding boxes and the area of their union, as can be seen in Figure 4.11.

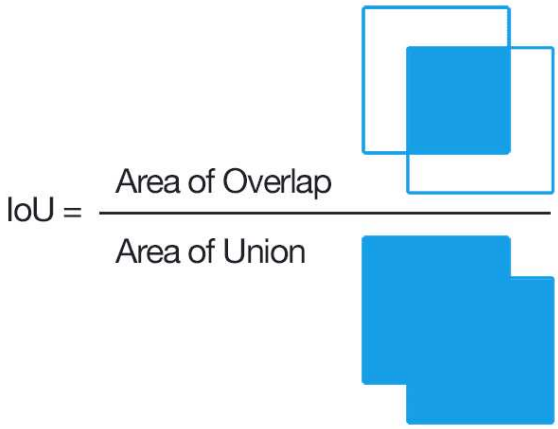

Figure 4.11: Geometric interpretation of the IoU metric between two rectangular bounding boxes.

The IoU metric will always generate a number between zero and one, so with a configurable threshold the user bounding boxes were then classified as either hits or misses. From this, each hit box was assigned to one of the four categories **true positive** (TP), **false positive** (FP), **false negative** (FN) and **true negative** (TN).

A **true positive** is counted if the IoU between a users bounding box and the ground truth is greater than the threshold, while **false positive** is counted if it is below for all ground truths. A **false negative** is counted if a ground truth bounding box is not matched towards any user bounding box. **true negative** is always zero as each frame is assumed to contain at least one ground truth bounding box.

From these four numbers three related statistics were calculated to decide the performance of the detection algorithm.

$$\mathrm{P} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FP}} \tag{4.3}$$

$$\mathrm{R} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FN}} \tag{4.4}$$

$$\mathrm{F1} = \frac{2 \cdot \mathrm{Precision} \cdot \mathrm{Recall}}{\mathrm{Precision} + \mathrm{Recall}} = \frac{\mathrm{TP}}{2 \cdot \mathrm{TP} + \mathrm{FP} + \mathrm{FN}} \tag{4.5}$$

Where P and R are precision and recall respectively. The precision metric measures the confidence of the detection algorithms in predicting bounding boxes, while the recall measures the ability of the algorithms to identify all targets. The F1 score is the harmonic mean between the two and is usually good to use to get a feel for both of the other metrics.

4.3.2 Open-loop evaluation

Open-loop testing is a method used to assess the performance of a system or device without any feedback control. In this setup, the input from the environment is applied to the application, and the output is observed without any corrective action. It provides a controlled environment to understand the basic behavior and characteristics of the system or for focused analysis of individual functionalities under specific inputs or conditions. Making it valuable as it enables early assessment of system components, allowing developers to identify issues before integration.

The evaluation process for open-loop testing is conducted on a per-signal basis. Let $T(t_t)$ represent the target signal, which is sampled from the `rec` file at discrete time instances t_t corresponding to each transmission. Similarly, let $S(t_s)$ denote the signal to be evaluated, sampled at discrete time instances t_s, corresponding to each output message from the application layer. As these two signals exist in distinct time domains, the target signal undergoes $T(t_t)$ linear interpolation $\hat{T}(t_s) = \mathcal{I}[T(t), t_t, t_s]$, to align with the timestamps of the signal to be evaluated, see Figure 4.12.

The two time series $S(t_s)$ and $\hat{T}(t_s)$ are then used to asses the performance of the system. Primarily through the use of the mean, max and standard deviation from the absolute difference of the time series.

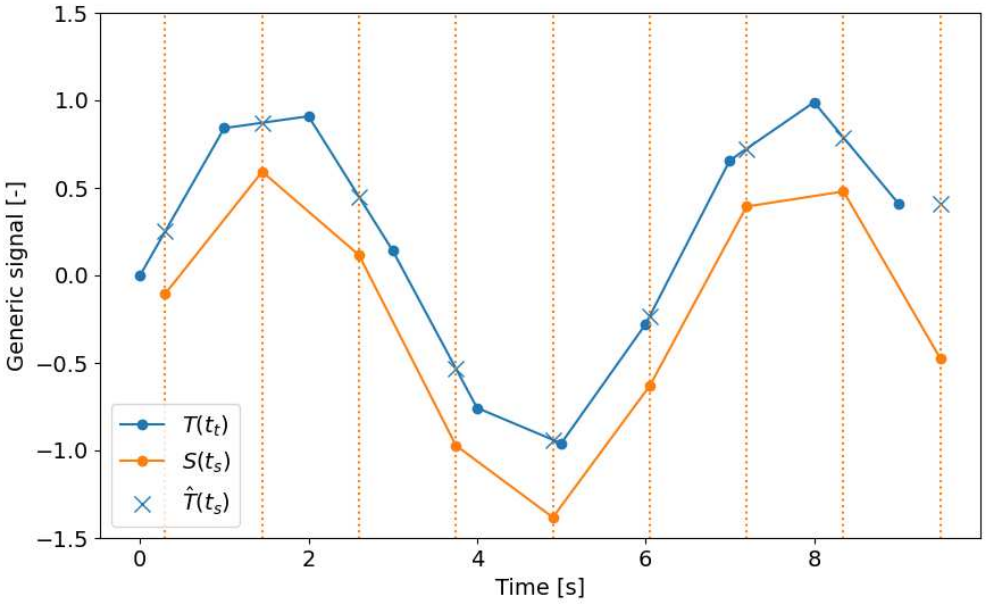

Figure 4.12: How the target signal $T(t_t)$ is converted into the interpolated target $\hat{T}(t_s)$ aligning with the signal $S(t_s)$ to be evaluated.

$$d_n = |S(n) - \hat{T}(n)| \text{ for } n \in t_s \tag{4.6}$$

$$S_{\text{mean}} = \frac{1}{N}\sum_{1}^{N} d_n \tag{4.7}$$

$$S_{\text{std}} = \sqrt{\frac{1}{N}\sum_{n=1}^{N}(d_n - \bar{d_n})^2} \tag{4.8}$$

$$S_{\text{max}} = \max\{d_1, d_2, \ldots, d_N\} \tag{4.9}$$

In the equation, N is the total number of captured signal values. These metrics is used to assess the performance of the application layer and can ether pass or fail the test depending on the magnitude of individual values.

4.3.3 Closed-loop evaluation

When running a closed-loop test the application layers output will affect the action layer, that will affect the environment layer that in turn will affect the application layer, creating a closed feedback loop. This can create a chaotic system were small changes in the application layers output can have large impacts on the future environment. Therefore, it is both impractical and difficult to decide on a good target signal, as was done with the open-loop evaluation in Section 4.3.2.

Instead, we adapted a goal oriented testing approach [29], were only the systems final state is evaluated, ignoring the immediate actions of the system. This means

that only the systems ability to achieve the given goals are evaluated, and not the strategy to achieve them. As an example, in the course TME290, the students are to develop an application layer capable of driving around a short track, constrained by blue and yellow cones on the sides.

Instead of defining the "best strategy" and forcing students to comply with it, only the two goals of reaching an imaginary finish line and avoiding going outside of the track limits are evaluated. The test is considered passed if the finish line is reached, and failed if the vehicle goes outside of the track limits. An additional soft goal of optimizing the time could also be set, but in the current implementation only hard goals are considered, as soft goals would require a ranking system to differentiate between solutions, something that was outside of our scope.

This gives rise to more diverse solutions, as all successful strategies are considered equal. If the same method as with open-loop testing would have been used, solutions could have started to converge towards the "best strategy" that was assigned as the target.

To solve the evaluation of closed-loop testing when driving around a track with the Kiwi car, both the goal of reaching the finish line and the one to avoid the track limits were implemented with the same microservice `object-detection` [24]. This microservice checks for collisions between a pre-configured bounding box of the vehicle that moves with the simulated position and other pre-defined and stationary objects. The goal of reaching the finish line was then considered passed when the bounding box of the Kiwi car collided with the finish line. And the other goal of not driving outside of the track limits was failed if the Kiwi car ever collided with imaginary walls between the cones on the track limits. In Figure 4.13 an example of both a failed and a passed test is shown.

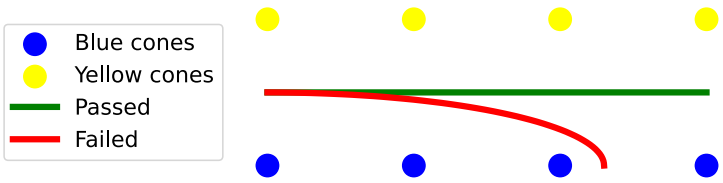

Figure 4.13: Example of the closed-loop evaluation. The red line is from a simulated vehicle that failed the test by leaving the allowed area, and the green line is from a simulated vehicle that reached the finish line without leaving the allowed area.

4.4 Test registry

In the DoDo framework, a test constitutes a combination of services that collectively construct the environment, action and evaluation layers. This environment can take the form of a simulation, real, or a hybrid of both. Additionally, each test requires

an application to undergo testing. This definition of a test enables reusability of microservices across multiple tests and test levels [11].

At this stage of the method, the individual services needed for a test have been developed and deployed. However, to operationalize these services into tests, they must be specified and packaged into an executable form. To achieve this, the `DoDo test registry` [21] was created. Currently, the registry houses nine tests created as part of this thesis.

The test registry organizes each test into a directory structure, each with files: `docker-compose.yml`, `.env`, `info.yml`, and a `README.md`. The packaging of tests is done within the docker-compose file. The additional files offer supplementary details and configurations for users and the CI/CD pipeline. They also enable easy adjustment of parameters within the docker-compose file.

This test structure renders each test self-contained, enabling its execution with only Docker installed. This makes for a independent and standalone test registry thats deployable and utilizable across diverse environments, including remote GitLab CI/CD pipelines.

4.5 Remote integration

Part of the purpose is to automate the testing procedure via GitLab. The general idea with an automated remote pipeline is to automate the *integration* and *deployment* of the code base, called CI/CD. The DoDo framework is a testing framework, hence it will focus on testing the code base, or the *continuous integration* part.

The goal is to validate the integration and functionality of the application microservices. To achieve this, the pipeline automatically initiates a new test each time code is committed to any of the microservices within the application. The testing workflow is illustrated in Figure 4.14.

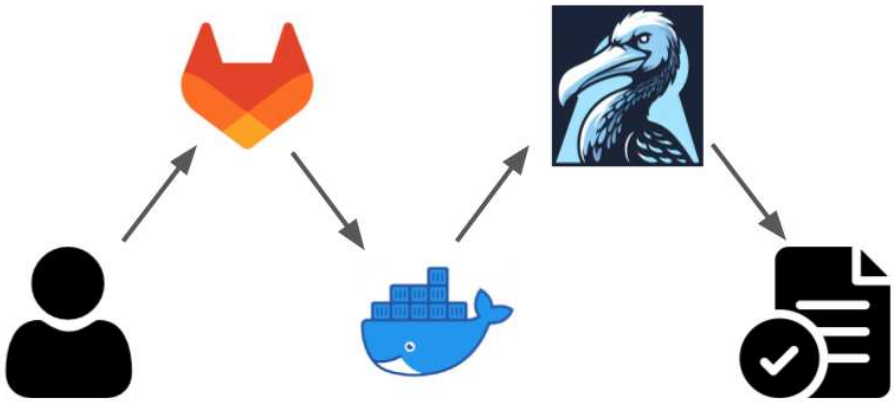

Figure 4.14: The frameworks CI/CD pipeline: (1) Push code to GitLab. (2) Build a Docker image. (3) Start end-to-end testing with the DoDo framework. (4) Present results.

Below, the methodology for remotely integrating tests from the `DoDo test registry` [21] achieved in two principal steps is outlined. First, it explores the configurations within GitLab. Second, the chapter examines the role of the DoDo runner software. For a comprehensive guide on configuring remote integration see the `CI/CD template` [30].

4.5.1 GitLab

The thesis integration of the `DoDo test registry` [21] into a CI/CD pipeline begins with configuring GitLab. The primary configurations needed for this include the repository structure and the use of GitLab runners and servers.

The repository structure followed one repository per application service, supplemented by a distinct deployment repository, as illustrated in Figure 4.15. This deployment repository manages the deployment of the application and the triggering of tests. It contains a docker-compose file to run the application and is configured to automatically trigger jobs whenever new code is committed to the default branch of any application service.

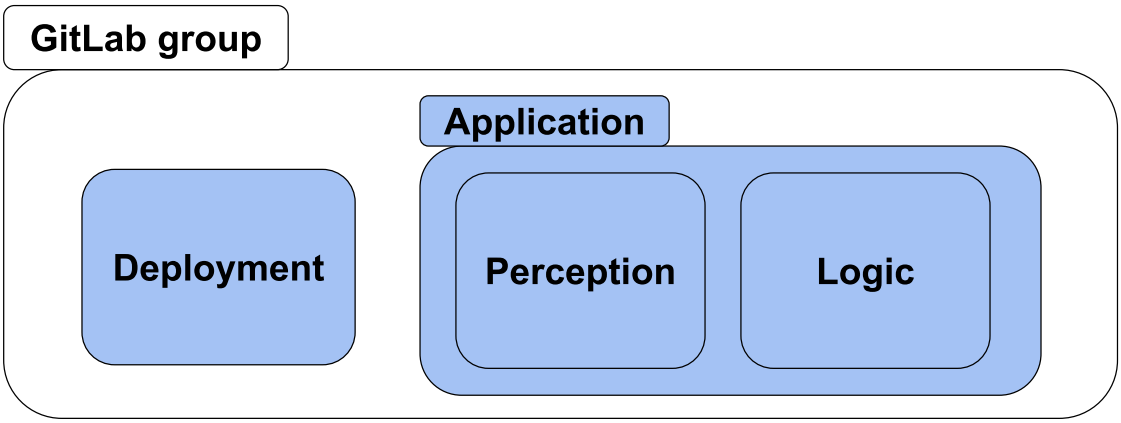

Figure 4.15: Proposed GitLab structure for integration of automated end-to-end testing with the DoDo framework.

This automatic triggering of jobs is achieved through a downstream job configuration in the application repositories. It is configured such that when code is merged into the default branch of an application service, GitLab builds a Docker image, adds it to the registry, and then triggers a downstream job in the deployment repository to start the testing procedures.

This approach enhances resource efficiency by using pre-built Docker images, which need only be pulled from the registry rather than built from scratch for each job. This significantly reduces the time and computational resources required for testing. Additionally, it ensures that the application uses the latest version of the code since jobs are triggered by the newest contributions from the service repositories, guaranteeing that the application is always tested against the latest changes.

The automatically triggered job in the deployment repository begins by connecting to a specified GitLab runner hosted on a server. This project uses three servers to

manage different aspects of the automated testing. Two servers are dedicated to the HIL rigs, while the third server is designated for the SIL and DIL tests.

The GitLab runners primary responsibilities include preparing the testing environment, such as logging into Docker, and initiate the DoDo runner. The DoDo runner is a software developed to execute tests in the `DoDo test registry` [21], capture the results and feed them back to the CI/CD pipeline. More details about the DoDo runner's features and its role in the CI/CD integration process will be provided in the next section.

4.5.2 DoDo runner

The `DoDo test runner` [22] is software developed to execute integration tests from the `DoDo test registry` [21] in a remote environment. Its role within the CI/CD pipeline begins once the job request reaches the server. Its main objective is to ensure a detailed and accurate interpretation of test results, a capability that the GitLab runner alone cannot provide. Moreover, it simplifies and enhances control over the testing process.

Developed in Python, the `DoDo test runner` [22] was installed using the instructions in the repository's `README.md` file. The GitLab runner initiates the Dodo runner through a shell command, which provides the necessary files and configurations. Essential components for running the Dodo Runner include the `docker-compose.yml` file for the application layer, a test registry with integration tests, and a YAML configuration file that specifies the tests to be conducted.

Upon initiation, the `DoDo test runner` [22] pulls the necessary Docker images and sets up the test suites as defined in the configuration file. The test suites contains one `docker-compose.yml` file per layer (environment, application, action and evaluation). These are started and the test is officially started once all services are running.

Test completion is ether from one any service finishing with an error code or all evaluation services finishing with no error code. Upon completion the test logs are parsed to distinguish between passed tests, failed tests (indicating that the test has a definable outcome but did not meet expected criteria), and errors.

The completion of tests occurs under two conditions: when any service exits with an error or when all evaluation services exits without errors. Following this, the test logs are analyzed to differentiate between passed tests, failed tests (where the test did not meet expected criteria despite a clear outcome), and errors in execution.

After the evaluation, a test report is generated and made available in GitLab. This report summarizes the results and offers insights into the health of the current code base, highlighting key metrics and data points to inform further development decisions.

5

Results

In this chapter, we present the results of this thesis. First, we will discuss the developed tests and trends related to the research questions. Next, we will detail the integration within the CI/CD pipeline, highlighting the current limitations. Finally, we will describe the two HIL rigs that were developed, emphasizing their capabilities and limitations.

5.1 Tests

This thesis developed tests for the TME290 Autonomous robots course at Chalmers University of Technology to address the research questions. The development resulted in nine tests, as shown in Table 5.1. Collectively, these tests rely on eleven microservices, detailed in Table 5.2.

Figure 5.1 illustrates the trend of the number of microservices, M relative to the number of tests, T. This trend seems to exhibit a concave behavior, as the number of microservices increases at a decreasing rate. Fitting the data to the function $M = aT^b$, yields $a = 0.66$ and $b = 0.78$. Importantly, the 95 % confidence interval for b is 0.63 to 0.92, meaning $b < 1$ with statistical significance. This outcome is attributed to the reusability of services, which results in reduced development time for new tests. The **microservices** column in Table 5.1 highlights this efficiency, demonstrating the sharing of several microservices across multiple tests.

The services and tests employed a consistent messaging standard across environments combined with Docker packaging, which resulted in seamless integration across platforms and environment levels. This meant that no additional work was required when transitioning between different environment levels or servers.

5.2 CI/CD integration

The tests developed based on the framework, presented in Section 5.1, were integrated with GitLab's CI/CD pipeline to automate the testing process. This integration was accomplished through the development of specialized software to execute the testing jobs on servers connected to the GitLab pipeline. This software, referred to as the DoDo runner and detailed in Section 4.5.2, also generates a post-test summary of the test results, which are presented on GitLab, as illustrated in Figure

Table 5.1: A table with all tests developed during the project.

Test name	Category	Description	Microservices
`sil-open-straight`	SIL	This is an open-loop software test with a straight track. It tests if the steering is reasonable.	Virtual camera, Beacon and Continuous-signal
`sil-closed-straight`	SIL	This is a closed-loop software test with a straight track. It tests if the software can drive the vehicle to the end of the track.	Virtual camera, Dynamic model, Integrator and Collision
`sil-open-left-curve`	SIL	This is an open-loop software test with a left-curved track. It tests if the steering is reasonable.	Virtual camera, Beacon and Continuous-signal
`sil-closed-left-curve`	SIL	This is a closed-loop software test with a left-curved track. It tests if the software can drive the vehicle to the end of the track.	Virtual camera, Dynamic model, Integrator and Collision
`sil-closed-kiwi-track`	SIL	This is a closed-loop software test with the official kiwi-track. It tests if the software can drive the vehicle to the end of the track.	Virtual camera, Dynamic model, Integrator and Collision
`dil-static-rig`	DIL	This is a test of the cone detection microservice with pre-recorded video. It tests how well cones are detected.	Rec replay and Continuous-signal
`dil-kiwi-track`	DIL	This is a test of the logic of the system with pre-recorded video. It tests if the steering is reasonable.	Rec replay and Continuous-signal
`hil-camera`	HIL	This is a test of the cone detection microservice with live video. It tests how well cones are detected.	Camera and Object detection
`hil-dynamic`	HIL	This is a test of the algorithmic logic of the system with live video. It tests if the software can drive the vehicle to the end of the track.	Camera, Dynamic model, Integrator, Gantry and Collision

Table 5.2: Description of microservices we used to compose the tests in Table 5.1.

Microservice	Description
Camera	The camera service takes images from the real camera and places them in shared memory.
Beacon	Simulates vehicle movement in open-loop tests by outputting OpenDLV messages with the current position at a specified frequency.
Virtual camera	Captures and renders scenes within a digital environment, mimicking the functionality of a physical camera.
Rec recorder	Records OpenDLV messages and camera feeds into a `rec` file format for playback and analysis.
Rec replay	Replays the contents of a `rec` file, including OpenDLV messages and camera feeds.
Continuous-signal	Tracks and evaluates a signal over time, comparing it against a target signal to ensure accuracy.
Collision	Detects and checks for collisions between two objects in the simulated environment.
Dynamic model	Simulates the dynamics of the Kiwi car.
Integrator	Numerically integrates the kinematic state of the Kiwi car into a position and orientation.
Gantry	Controls the position of the dynamic HIL rig through serial communication with an Arduino.
Object detection	Evaluates the object detection algorithm in the application layer.

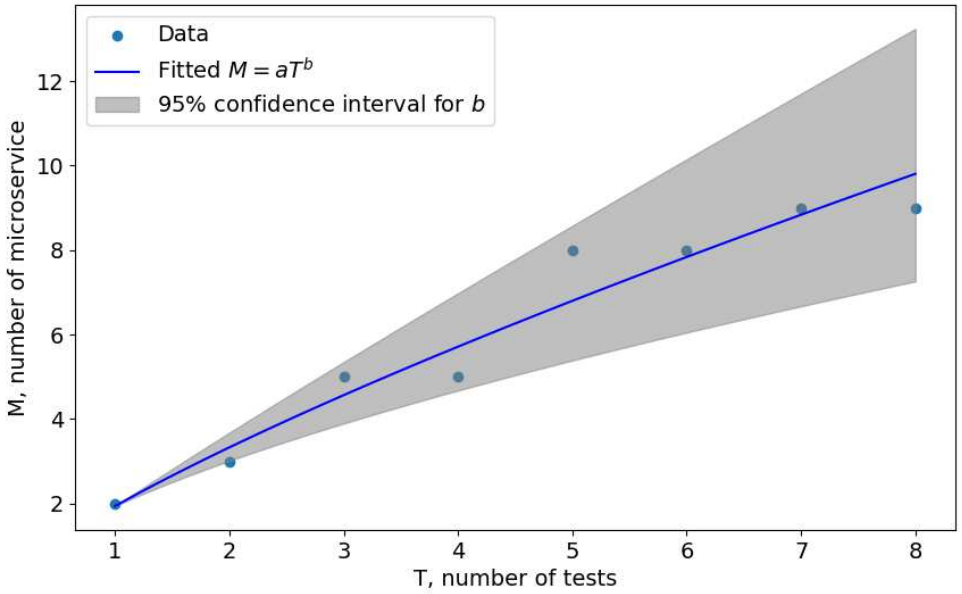

Figure 5.1: Cumulative amount of microservices, M, required to develop T tests. The trend could indicate that as more tests are developed fewer and fewer microservices must be developed per test as the tests can share previous microservices.

5.2.

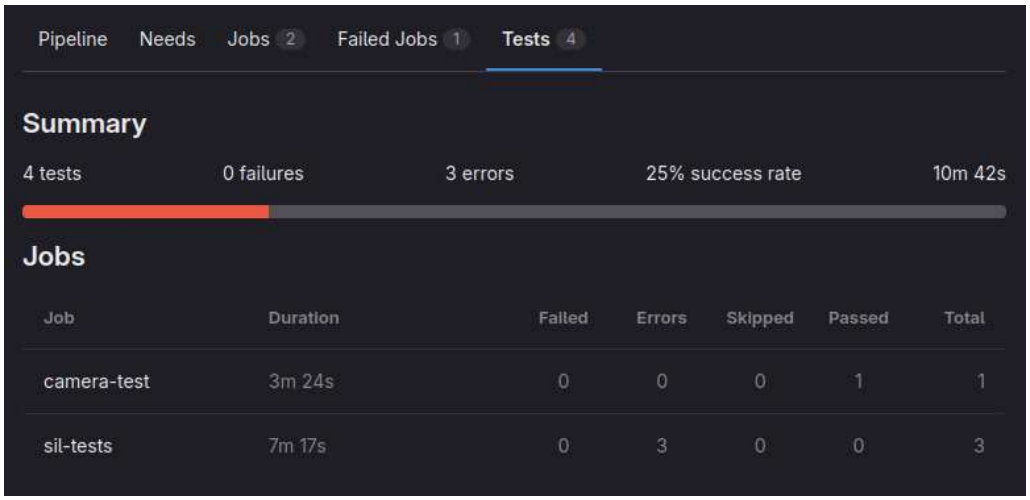

Figure 5.2: Test summary reported in GitLab, summarized by the DoDo runner.

However, a significant limitation of the CI/CD integration is its restriction to single-user connectivity per runner instance at any given time. This constraint arises from the use of UDP multicast for inter-service communication, which broadcasts messages to all nodes within the network. Consequently, concurrent tests results in communication conflicts.

This parallelization limitation results in delays in the CI/CD pipeline when several jobs start simultaneously. This problem is further compounded by the relatively long duration required to run each test. With the two primary contributors being the suite setup and the test execution.

The execution time for a test is reasonable and fairly consistent, typically ranging between ten and sixty seconds. In contrast, the setup time varies significant across different devices, as illustrated in Figure 5.3. The primary factor influencing the setup time is the Docker image extraction phase, which occurs during the pulling. Ultimately, this might introduce significant delay to the CI/CD pipeline.

This extraction process was observed to be considerably slower on the Raspberry Pi devices used in this thesis compared to those in the Kiwi car. The underlying cause of this discrepancy remains unclear, as optimizing this process was not a primary focus of this thesis.

5.3 Hardware-in-the-loop

In this section, we present the final construction, functionality, and limitations of both the static and dynamic HIL rigs. These rigs are crucial for enabling HIL testing. They serve to validate both the perception and logic capabilities of the application layer in controlled and dynamic environments. Detailed descriptions, design considerations, and operational insights for each rig are provided to highlight their contributions to the overall testing framework.

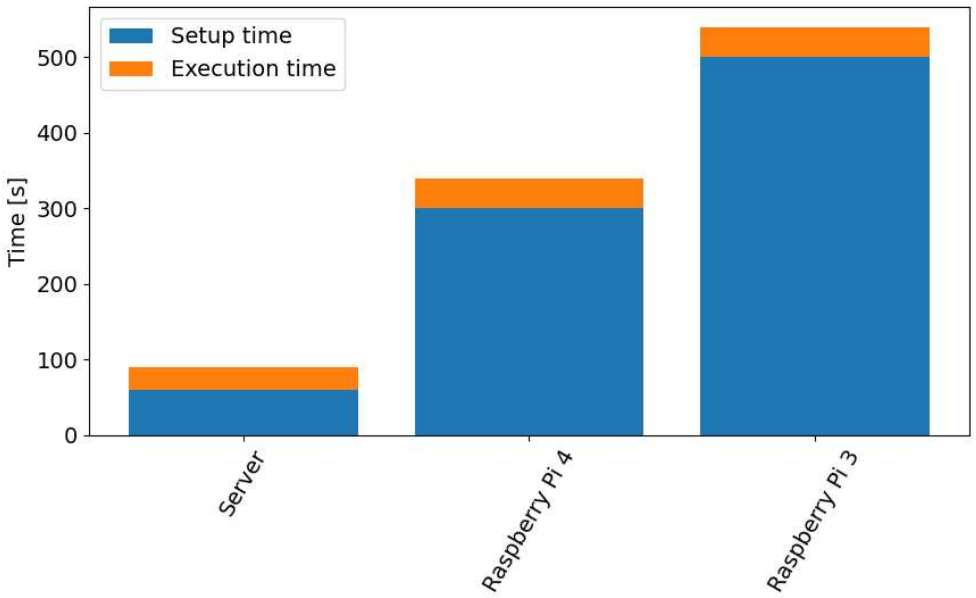

Figure 5.3: Total test time, split between the setup time and execution time.

5.3.1 Static HIL rig

The static HIL rig was specifically designed to test the perception capabilities of the application layer in a controlled environment. The final construction of the static HIL rig is shown in Figure 5.4, and its point of view is illustrated in Figure 5.5. Given that the perception of the application layer heavily depends on color accuracy, high image quality was prioritized over frame rate, resulting in an exposure time of 100 ms, or a frame rate limit of 10 Hz.

Figure 5.4: The final construction and placement of the static HIL rig.

Figure 5.5: Point of view for the camera.

The rig occupies a compact footprint of 60 × 70 cm. The primary requirements for its operation are access to a power outlet and an internet connection. The

combination of its small size and minimal cabling makes the static HIL rig highly portable, facilitating easy relocation to different environments for varied testing scenarios. This mobility enabled the efficient generation of labeled datasets from different environments, serving as a valuable secondary function of the rig. Examples of four different environments can be seen in Figure 5.6.

(a) Normal conditions.

(b) Background clutter.

(c) Bright conditions.

(d) Dark conditions.

Figure 5.6: Static HIL rig camera snapshot for four different environments.

5.3.2 Dynamic HIL rig

The dynamic HIL rig was specifically designed to test the logic capabilities of the application layer in a controlled environment. It was constructed based on the specifications outlined in Section 4.2 and the final construction is shown in Figure 5.7. The rig occupies a footprint of $220 \times 170\,\text{cm}$ with a workspace of $190 \times 140\,\text{cm}$ for translational movement and $-170°$ to $170°$ for rotational movement.

This setup provides the camera and the vehicle sensors with adequate space to navigate the prepared track. However, the rig is constrained to translational accelerations of $0.5\,\text{m}\,\text{s}^{-2}$ and translational speeds of $1\,\text{m}\,\text{s}^{-1}$, which are slower than the top speed of a real Kiwi car. The rotational movement is also limited, with an angular acceleration constraint of $1\,\text{rad}\,\text{s}^{-2}$ and an angular velocity constraint of $1\,\text{rad}\,\text{s}^{-1}$. These rotational constraints do not impact the test as the vehicle would need to operate at speeds well beyond the rig's limitations to reach such rotational extremes.

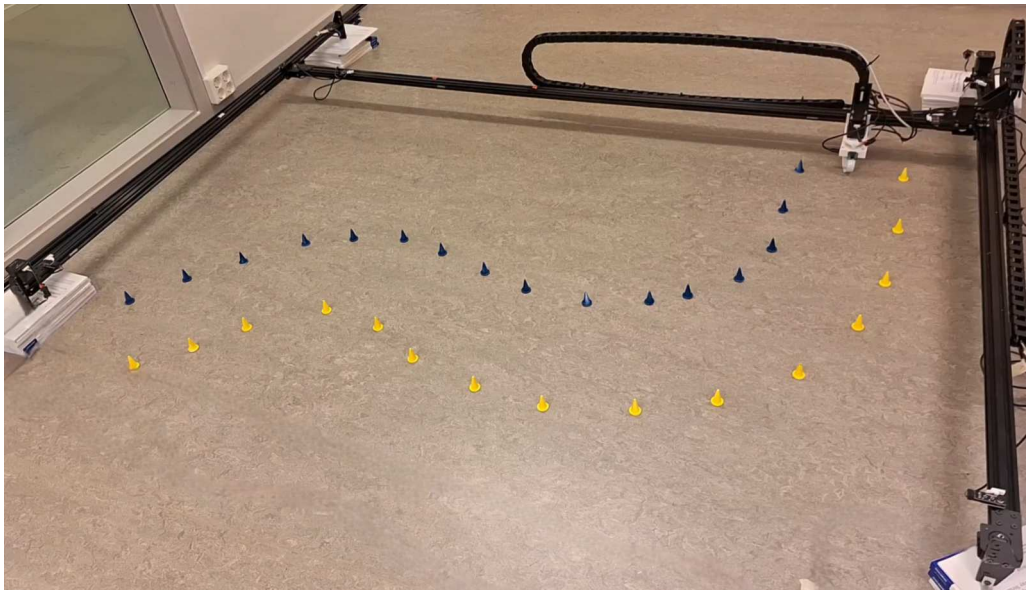

Figure 5.7: Final construction on the dynamic HIL rig.

6

Discussion

This section presents a discussion of the findings in this thesis. We begin by interpreting the results and exploring their implications. This is followed by an examination of the limitations of our findings and methodology. Finally, we offer suggestions for future research and conclude with a summary of our work.

6.1 Interpretation of results

With the successful development of several tests, presented in Table 5.1, this thesis have shown that a MSA supports the development of tests across multiple levels, including SIL, DIL and HIL. With the usage of Docker to deploy microservices, and OpenDLV to facilitate communication between them, the transition between test levels have been seamless, time efficient, and without any major hurdles.

This was particularly noticeable after the completion of or HIL rigs. With SIL and DIL testing already setup, the extension to HIL testing required minimal effort. As can be seen in Table 5.1 the only software development required to facilitate HIL testing was microservices to interact with the hardware. Both the simulation, and evaluation could be brought over from previously developed SIL and DIL tests. This highlights the efficiency of the MSA to extend testing to new areas, with minimal additional effort.

Furthermore, the results shown in Figure 5.1 indicate that a MSA used during testing could ultimately lead to reduced resources required for testing. Our data shows that the exponent $b < 1$, and therefore, fewer microservices must be developed for each new test case. This could lead to reduced time and cost spent on testing if a MSA is implemented early during development.

In Section 5.2 we presented our successful implementation of a CI/CD pipeline on GitLab with our tests. This shows that it is feasible to integrate tests with a MSA in a CI/CD pipeline, but a lack of dedicated software made it more challenging than it needs to be. The lack of software made us use develop the DoDo runner to setup, execute, evaluate and tear-down each test. This software should not be seen as a solution to the problem as it is quite biased towards our use case, but it could serve as inspiration for future efforts to develop generic solutions for MSA testing.

6.2 Implications

In Section 2.2, we discussed previous work done within the field of MSA testing. Or own findings about the limitations and advantages with MSA testing is in agreement with these studies. Mainly these three problems: (1) There is currently a lack of dedicated software to facilitate testing with MSA applications, (2) it is well suited for both HIL and multi-level testing and (3) it is challenging to integrate it with CI/CD pipelines to automate testing.

Our results and previous studies therefore agree that more work on software to facilitate MSA testing is required. In this thesis we developed the DoDo runner to facilitate our tests and to partially solve problem (1) above. As shown in this thesis, solving problem (1) is a big step towards solving problem (3) as a large part of our work with the CI/CD integration was the DoDo runner.

6.3 Constraints and considerations

Throughout the thesis, several constraints were imposed due to both time limitations and the specific requirements of the TME290 Autonomous robots course. These constraints influenced our software choices and the development of test cases.

The course dictated our choices in four key areas: programming language, containerization methods, communication protocol, and Git web service provider. We used C++ and Python to meet course requirements, Docker was selected for the containerization of microservices, while OpenDLV was chosen as the communication protocol for interfacing with the Kiwi car. GitLab was used as the Git web service provider, which facilitated HIL testing by allowing easy setup of a runner directly on the HIL rigs, streamlining CI/CD integration.

These technical choices were appropriate for our needs, and while we do not claim superiority over other software options, they had no significant impact on the results.

The test cases were developed with the course in mind, which introduces a bias as they are tailored to the Kiwi car—a system designed with a microservices architecture. Consequently, the testing framework integrates seamlessly with this system. Other applications might face challenges when integrating with the same framework.

A significant limitation of the current implementation of the DoDo runner is its inability to run multiple jobs in parallel. This limitation can significantly increase the time between a code push and the presentation of results, especially when multiple users are working simultaneously, which is likely near course deadlines. Enhancing the DoDo runner to support parallel execution would greatly benefit the course.

The parallelization issue arises from the use of the host machine's internal network for communication between microservices, leading to potential interference when multiple tests run simultaneously. A potential solution is to use an isolated network for each test, configurable through Docker, though this would require per-test modifications to the user's `docker-compose.yml` file. Alternatively, running each test

on a separate multicast address, configurable in OpenDLV, could address this issue, but it also requires per-test adjustments.

The project was intended to align with course objectives and gather student feedback. However, due to time constraints, the evaluation was only partially completed. Despite limited interactions, the feedback received suggests that a framework like ours would be well-received. Although the course evaluation included questions about the thesis, the responses were not available at the time of writing this report.

6.4 Future work

As stated in Section 6.2 we see a need for more work in developing dedicated MSA testing software. Our work with the DoDo runner in this thesis is a starting point but far from a general solution to the problem. Currently it has several drawbacks, such as only only supporting one test at a time and a hard to learn configuration. A general solution should support parallelization as so that simulations can be run in parallel to reduce time. An easy interface also reduces the technical knowledge required to use the tool.

Our work with the HIL rig could also see some upgrades. The biggest limitation at the moment is that it does not support 360° movement, limiting its use cases to tasks that do not require this. An upgrade that supports 360° movement would increase the possible test cases on the rig.

It would also be very beneficial to get user feedback on the system. In upcoming iterations of the TME290 Autonomous robots course, our work can hopefully be used in the course from the start, enabling user feed back from next years students.

6.5 Conclusion

In this thesis, we have investigated the usage of a MSA to test software. We developed several test cases, across multiple test levels, including SIL, DIL, and HIL, and successfully integrated them with a CI/CD pipeline. This has shown the feasibility of a MSA for test development.

Furthermore, we conclude that a MSA is well suited for multi-level testing, as microservices can easily be shared between test levels, reducing the amount of duplicate code. One downside however, is the added complexity of CI/CD integration. In this thesis, we developed the DoDo-runner to overcome this challenge, but with future efforts dedicated towards developing a better and more general solution, the CI/CD integration could be significantly simplified.

Importantly we also develop two functional HIL rigs that can be used to test the application layer on the Kiwi car. This could generate huge benefits for the course TME290 as students in the course could test their software in a controlled and standardized environment, reducing time and effort spent on testing.

As a final conclusion to this thesis we want to encourage everyone to continue to DoDo testing!

Bibliography

[1] *Cyber–physical system*. en. Page Version ID: 1222053961. May 2024. URL: https://en.wikipedia.org/w/index.php?title=Cyber%E2%80%93physical_system&oldid=1222053961 (visited on 05/20/2024).

[2] Volkan Gunes et al. "A Survey on Concepts, Applications, and Challenges in Cyber-Physical Systems". eng. In: *KSII Transactions on Internet and Information Systems (TIIS)* 8.12 (2014). Publisher: Korean Society for Internet Information, pp. 4242–4268. ISSN: 1976-7277. DOI: 10.3837/tiis.2014.12.001. URL: https://koreascience.kr/article/JAKO201403760397435.page (visited on 05/20/2024).

[3] Cynthia Rudin. *Stop Explaining Black Box Machine Learning Models for High Stakes Decisions and Use Interpretable Models Instead*. arXiv:1811.10154 [cs, stat]. Sept. 2019. DOI: 10.48550/arXiv.1811.10154. URL: http://arxiv.org/abs/1811.10154 (visited on 05/20/2024).

[4] Ronghua Xu et al. *BlendMAS: A BLockchain-ENabled Decentralized Microservices Architecture for Smart Public Safety*. arXiv:1902.10567 [cs]. Feb. 2019. DOI: 10.48550/arXiv.1902.10567. URL: http://arxiv.org/abs/1902.10567 (visited on 05/20/2024).

[5] Kleanthis Thramboulidis, Danai C. Vachtsevanou, and Alexandros Solanos. *Cyber-Physical Microservices: An IoT-based Framework for Manufacturing Systems*. arXiv:1801.10340 [cs]. Apr. 2018. DOI: 10.48550/arXiv.1801.10340. URL: http://arxiv.org/abs/1801.10340 (visited on 05/20/2024).

[6] Christian Berger, Bjornborg Nguyen, and Ola Benderius. "Containerized Development and Microservices for Self-Driving Vehicles: Experiences & Best Practices". en. In: *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*. Gothenburg, Sweden: IEEE, Apr. 2017, pp. 7–12. ISBN: 978-1-5090-4793-2. DOI: 10.1109/ICSAW.2017.56. URL: http://ieeexplore.ieee.org/document/7958428/ (visited on 11/09/2023).

[7] Atlassian. *Microservices Architecture*. en. URL: https://www.atlassian.com/microservices/microservices-architecture (visited on 05/21/2024).

[8] Muhammad Waseem et al. "Testing Microservices Architecture-Based Applications: A Systematic Mapping Study". In: *2020 27th Asia-Pacific Software Engineering Conference (APSEC)*. ISSN: 2640-0715. Dec. 2020, pp. 119–128. DOI: 10.1109/APSEC51365.2020.00020. URL: https://ieeexplore.ieee.org/abstract/document/9359275 (visited on 02/15/2024).

[9] Nuha Alshuqayran, Nour Ali, and Roger Evans. "A Systematic Mapping Study in Microservice Architecture". In: *2016 IEEE 9th International Conference on*

Service-Oriented Computing and Applications (SOCA). Nov. 2016, pp. 44–51. DOI: 10.1109/SOCA.2016.15. URL: https://ieeexplore.ieee.org/document/7796008 (visited on 02/15/2024).

[10] Davide Taibi, Valentina Lenarduzzi, and Claus Pahl. "Architectural Patterns for Microservices: A Systematic Mapping Study". In: Mar. 2018. DOI: 10.5220/0006798302210232.

[11] Iñigo Aldalur et al. "A microservice-based framework for multi-level testing of cyber-physical systems". en. In: *Software Quality Journal* 32.1 (Mar. 2024), pp. 193–223. ISSN: 1573-1367. DOI: 10.1007/s11219-023-09639-z. URL: https://doi.org/10.1007/s11219-023-09639-z (visited on 02/14/2024).

[12] *OpenDLV—A modular and scalable software framework for autonomous systems and robots*. URL: https://opendlv.org/ (visited on 01/26/2024).

[13] *testing / opendlv-tutorial-kiwi · GitLab*. en. Feb. 2022. URL: https://git.opendlv.org/testing/opendlv-tutorial-kiwi (visited on 06/18/2024).

[14] *Our research platforms and projects*. en. Dec. 2022. URL: https://www.chalmers.se/en/infrastructure/revere/research/ (visited on 04/24/2024).

[15] *Microservices*. en. Page Version ID: 1217031192. Apr. 2024. URL: https://en.wikipedia.org/w/index.php?title=Microservices&oldid=1217031192 (visited on 04/22/2024).

[16] *Docker overview*. en. 200. URL: https://docs.docker.com/get-started/overview/ (visited on 04/23/2024).

[17] *Docker (software)*. en. Page Version ID: 1218327104. Apr. 2024. URL: https://en.wikipedia.org/w/index.php?title=Docker_(software)&oldid=1218327104 (visited on 04/23/2024).

[18] *What is a container?* en. 800. URL: https://docs.docker.com/guides/walkthroughs/what-is-a-container/ (visited on 04/23/2024).

[19] *Docker Compose overview*. en. 100. URL: https://docs.docker.com/compose/ (visited on 04/23/2024).

[20] Christian Berger. *chrberger/libcluon*. original-date: 2017-12-18T22:15:54Z. Mar. 2024. URL: https://github.com/chrberger/libcluon (visited on 03/28/2024).

[21] *Chalmers Revere / Dodo / runner / DODO-test-registry · GitLab*. en. Apr. 2024. URL: https://git.chalmers.se/chalmers-revere/dodo/runner/dodo-test-registry (visited on 06/01/2024).

[22] *Chalmers Revere / Dodo / runner / DODO-test-runner · GitLab*. en. Apr. 2024. URL: https://git.chalmers.se/chalmers-revere/dodo/runner/dodo-test-runner (visited on 06/01/2024).

[23] *community / opendlv-virtual-camera · GitLab*. en. Jan. 2024. URL: https://git.opendlv.org/community/opendlv-virtual-camera (visited on 06/01/2024).

[24] *Chalmers Revere / Dodo / evaluation / camera · GitLab*. en. Mar. 2024. URL: https://git.chalmers.se/chalmers-revere/dodo/evaluation/camera (visited on 06/01/2024).

[25] *testing / opendlv-sim-global · GitLab*. en. May 2024. URL: https://git.opendlv.org/testing/opendlv-sim-global (visited on 06/01/2024).

[26] *community / opendlv-virtual-space · GitLab*. en. May 2023. URL: https://git.opendlv.org/community/opendlv-virtual-space (visited on 06/01/2024).

[27] *Chalmers Revere / Dodo / hardware / HIL-Kiwi-Car · GitLab*. en. Jan. 2024. URL: https://git.chalmers.se/chalmers-revere/dodo/hardware/HIL-Kiwi-Car (visited on 06/01/2024).

[28] Md Atiqur Rahman and Yang Wang. "Optimizing Intersection-Over-Union in Deep Neural Networks for Image Segmentation". en. In: *Advances in Visual Computing*. Ed. by George Bebis et al. Cham: Springer International Publishing, 2016, pp. 234–244. ISBN: 978-3-319-50835-1. DOI: 10.1007/978-3-319-50835-1_22.

[29] Duy Cu Nguyen, Anna Perini, and Paolo Tonella. "A Goal-Oriented Software Testing Methodology". en. In: *Agent-Oriented Software Engineering VIII: 8th International Workshop, AOSE 2007, Honolulu, HI, USA, May 14, 2007, Revised Selected Papers*. Ed. by Michael Luck and Lin Padgham. Berlin, Heidelberg: Springer, 2008, pp. 58–72. ISBN: 978-3-540-79488-2. DOI: 10.1007/978-3-540-79488-2_5. URL: https://doi.org/10.1007/978-3-540-79488-2_5 (visited on 05/15/2024).

[30] *Chalmers Revere / Dodo / dodo-templates / ci-cd-template · GitLab*. en. Apr. 2024. URL: https://git.chalmers.se/chalmers-revere/dodo/dodo-templates/ci-cd-template (visited on 06/01/2024).

DEPARTMENT OF SOME SUBJECT OR TECHNOLOGY
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden
www.chalmers.se

CHALMERS
UNIVERSITY OF TECHNOLOGY