Institute of Architecture of Application Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Automating Deployment and Testing in Distributed Networks of Electronic Control Units

Roman Vintonyak

Course of Study: Software Engineering

Examiner: Prof. Dr. Marco Aiello

Supervisor: Dr. Pascal Hirmer

Commenced: October 22, 2024

Completed: April 22, 2025

Abstract

Increasing complexity in automotive software has made manual testing and deployment in distributed networks of Electronic Control Units (ECUs) both time-consuming and error-prone. This thesis explores and implements a framework automating deployment and testing in distributed networks of ECUs. The proposed solution combines Ansible-based deployment with Gherkin-style test case descriptions, integrated into a CI/CD pipeline to enable consistent and repeatable testing automation. The resulting prototype called Automated Deployment and Testing of ECUs (ADATE), automates the testing of software components with automated deployment across simulated devices in a distributed environment. The framework demonstrates how automation in a distributed environment can make the testing process of ECUs both more efficient and reduce manual effort, offering a foundation for future adaptations in real-world automotive environments.

Contents

List of Figures

List of Tables

List of Listings

List of Abbreviations

ADATE Automated Deployment And Testing of ECUs. 44

AI Artificial Intelligence. 20

API Application Programming Interface. 32

CAN Controller Area Network. 35

CDPS Child Detection Protection Scenario. 48

CI Continuous Integration. 44

CI/CD Continuous Integration / Continuous Deployment. 16

DSL Domain Specific Language. 30

DUT Device Under Test. 36

ECU Electronic Control Unit. 15

Euro NCAP European New Car Assessment Program. 53

GUI Graphical User Interface. 37

HTML Hypertext Markup Language. 38

HTTP Hypertext Transfer Protocol. 57

IoT Internet of Things. 20

IP Internet Protocol. 48

JSON JavaScript Object Notation. 26

KPI Key Performance Indicator. 17

LAN Local Area Network. 38

MQTT Message Queue Telemetry Transport. 24

Nmap Network Mapper. 55

OEM Original Equipment Manufacturer. 15

OS Operating System. 22

OTA Over-the-Air. 61

RDF Ressource Description Framework. 33

RQ Research Question. 16

RTT Round-Trip Time. 56

SQ Search Queries. 19

SSD Semantic Service Description. 32

SSH Secure Shell. 29

TOSCA Topology and Orchestration Specification for Cloud Applications. 33

UDS Unified Diagnostic Services. 36

URL Uniform Resource Locator. 63

UUID Universally Unique Identifier. 60

WLAN Wireless Local Area Network. 49

YAML YAML Ain't Markup Language. 29

1 Introduction

In recent decades, the number and complexity of Electronic Control Units (ECUs) in modern vehicles have increased significantly. Today, modern cars can contain over 100 ECUs on average, each responsible for implementing critical functionalities such as powertrain control, infotainment, safety features, and more [MTK+25]. Given their essential role in the overall vehicle system ensuring their reliability and security is essential. Consequently, thorough testing of ECUs has become an indispensable part of the development process for automotive Original Equipment Manufacturers (OEMs).

Despite advances in testing environments, current testing including deployment of software components and test execution remains a very resource-intensive, error-prone, and time-consuming process, in both traditional centralized, or virtualized distributed testing environments [BD10; HJL15; MB20]. Furthermore, integration testing often occurs at a late stage in the V-model development cycle, reducing opportunities for early feedback and increasing the cost of fixing integration issues [LZZ16].

There are existing approaches for automating deployment and testing, in some cases even within distributed environments, albeit to a limited extent. However, a unified workflow that seamlessly combines automated deployment and testing is still lacking, particularly in the context of the automotive domain.

Taken together, these challenges highlight a significant research and engineering gap: the need for an automated, distributed, and decentralized testing framework tailored for heterogeneous ECU networks, incorporating both automated deployment and testing into one seamless workflow. Such a framework would not only accelerate the development process, but also improve test reliability, reduce human error, and support geographically distributed test execution.

This thesis aims to address this gap by exploring state-of-the-art approaches and technologies for automated and distributed deployment and testing within heterogeneous environments of ECUs. Building on the insights gathered, a framework is proposed and implemented to enable fully automated and distributed deployment and testing processes.

To systematically address this gap and better understand the underlying requirements and limitations, this thesis is guided by the following research questions:

RQ1: What are the requirements for test automation in a distributed and decentralized automotive environment?

RQ2: How can we achieve fully automated test execution through deployment and CI/CD pipeline integration?

RQ3: How can we evaluate the effectiveness of an automated approach compared to the state-of-the-art, e.g., in a non-or semi-automated setup?

To answer **RQ1**, a literature review was conducted to identify current limitations and fundamental requirements for distributed deployment and testing. The findings reveal a lack of standardization, limited flexibility in deployment tools, and substantial challenges integrating these solutions into Continuous Integration / Continuous Deployment (CI/CD) workflows [BD10; HJL15; LDP22].

Ultimately, the evidence indicates, that currently there is a high demand of automation processes for the development workflow of ECUs, as their complexity grows significantly over the last decades, specifically in the automotive sector. Incorporating distribution into testing is equally essential, as it enables decentralized execution across geographically distributed environments, increasing the robustness and efficiency of ECU testing.

These issues can be addressed through automation processes, not only removing the human factor from the entire process but also enabling significantly more efficient development. Accordingly, the automotive industry could significantly reduce costs by automating the entire development process of ECUs, with a particular focus on testing.

To address **RQ2**, this work introduces a framework that leverages a decentralized and distributed network of ECUs to enable automated testing, while ensuring compliance with the fundamental requirements identified in Research Question (RQ)1 during the literature review. By automating the deployment and testing processes, the framework supports a faster and more scalable development cycle for ECU-based systems.

The development process started with first establishing a high-level approach for the framework, based on the systematic literature review, incorporating fundamental requirements. This conceptual approach enables future work to use it as a foundation for developing other frameworks addressing similar challenges.

Our framework must adhere to three key principles to implement the aforementioned solution. First, it needs to be distributed, allowing for the testing of ECUs to be conducted on a global scale. This ensures a flexible development process that is no longer constrained by geographical boundaries. Additionally, the framework must be decentralized, enabling peer-to-peer communication between devices. This facilitates the seamless integration of heterogeneous systems and enhances both scalability and adaptability. Lastly, automation is crucial to eliminate error-prone human intervention, thereby streamlining the development workflow and improving overall system reliability.

To address RQ3, this work proposes a set of four Key Performance Indicators (KPIs) to evaluate the applicability of the frame-work in a real-world scenario. Our proposed evaluation approach includes Time, Fault Detection, Cost, and Manual Intervention, drawing on insights from literature focused on the evaluation of testing or deployment automation processes. This enables a comprehensive assessment in accordance with established scientific standards.

In summary, this thesis identifies key requirements for automated testing in distributed automotive environments, proposes a conceptual framework built around distribution, decentralization, and automation, implements a prototype, and outlines an evaluation strategy based on defined KPIs. By addressing these aspects, this thesis provides a solid foundation for advancing scalable and efficient ECU testing workflows in future research and real-world applications.

2 Literature Review

2.1 Search Strategy

This part of the thesis presents all methods and approaches used to fulfill and answer the RQs and establish a comprehensive set of related works. To achieve this, we utilized a systematic literature review approach. This thesis's initial collection of relevant works began with applying a keyword-based search strategy [BDA+99; KC07]. The literature review was conducted in two stages, each corresponding to specific RQs.

First, a thematic approach was employed to structure the foundation of this work by addressing RQ1: "What are the requirements for test automation in a distributed and decentralized automotive environment?". This review can be structured into three primary literature domains: Distributed and Decentralized Networks, Deployment in Distributed and Decentralized Networks, and Testing in a Distributed and Decentralized Network. Given this, the following two academic search engines are utilized: ACM Digital Library and IEEE Xplore.

Addressing RQ1 following Search Queries (SQ) where applied:

- SQ1.1: ("deployment" AND ("iot application" OR "distributed application") AND ("automated" OR "decentralized" OR "network"))
- SQ1.2: (("distributed network" OR "distributed system") AND "decentralized" AND ("computing")) NOT "blockchain"
- SQ1.3: "automated testing" AND "ECUs"
- SQ1.4: ("test case design" OR "test case specification" OR "testing frameworks") AND ("distributed systems" OR "IoT devices" OR "automotive systems")

After the initial search, further screening was conducted by reading the abstracts and assessing the relevance of the literature regarding this thesis. The resulting findings of this literature review, including the number of relevant works identified in each domain, are summarized in Table 2.1, resulting in a total of 2312.

These findings were then screened by title and abstract, resulting in 35 highly relevant research works. Inclusion and exclusion criteria were applied during the screening process, with a focus on literature related to distributed networks, Internet of Things (IoT) applications, decentralized environments, and the automotive context. Conversely, publications not in English or those addressing Artificial Intelligence (AI) and blockchain topics were deemed irrelevant for further screening, with the exception of one German publication that was included due to its high relevance.

Search Query	ACM Digital Library		IEEE Xplore	
	Pre Screening	Post Screening	Pre Screening	Post Screening
SQ1.1:	1221	17	60	5
SQ1.2:	648	4	73	5
SQ1.3:	39	0	23	1
SQ1.4:	239	3	9	0

Table 2.1: Result of Systematic Literature Review: RQ1

Next, we conducted a focused literature review to address RQ3: "How can we evaluate the effectiveness of an automated approach compared to the state-of-the-art, e.g., in a non-or semi-automated setup?". This stage concentrated on identifying state-of-the-art evaluation methods and benchmarking criteria for assessing automated versus traditional testing strategies. The following SQs were utilized to search for and identify relevant literature regarding RQ3 systematically:

- SQ3.1: ("test automation" OR "automated testing") AND ("manual testing" OR "semi-automated") AND ("evaluation" OR "comparison" OR "effectiveness")
- SQ3.2: ("deployment automation" OR "automated deployment") AND ("manual deployment" OR "semi-automated deployment") AND ("evaluation" OR "comparison" OR "effectiveness")

Analogous to the results discussed concerning the systematic research for RQ2, the findings for RQ3 are likewise visualized in Table 2.2. Ultimately, these findings resulted in 785 studies and 21 following the screening of titles and abstracts.

Search Query	ACM Digital Library		IEEE Xplore	
	Pre Screening	Post Screening	Pre Screening	Post Screening
SQ3.1:	554	5	191	12
SQ3.2:	38	3	2	1

Table 2.2: Result of Systematic Literature Review: RQ2

In summary, this approach provides a comprehensive overview of current state-of-the-art research on distributed networks, deployment, and testing, with particular emphasis on applications within the automotive sector and potential evaluation methods for assessing the effectiveness of automated approaches in comparison to state-of-the-art solutions. Beyond the systematic keyword search, additional relevant literature was identified through other credible sources.

2.2 Foundational Research

This chapter explores foundational research regarding distributed networks as well as deployment and testing, focusing on their fundamental aspects relevant to this thesis. It establishes a comprehensive foundation for the methodologies, approaches, and key technologies that support the conceptual approach and prototypical implementation of this thesis.

2.2.1 Distributed and Decentralized Network

Distributed and decentralized networks play a crucial role in modern systems, particularly in fields such as the Internet of Things (IoT) and automotive applications. A distributed and decentralized network is a collection of diverse, independent, and autonomously working components capable of interacting and collaborating to achieve one shared objective [DG08; Lam19; Mar21; TS07; ZZT10]. These components of a distributed network can consist of physical or simulated devices in the form of sensors or actuators, especially in the field of IoT [Mar21; TS07].

Within the context of this thesis, it is important to distinguish between the terms 'distributed' and 'decentralized' as both concepts, although related, are used to describe two different aspects of system structure. A distributed system is a structure in which components are geographically separated and communicate over a network to achieve a common goal. Distribution is concerned with physical or logical distribution of elements. On the other hand, decentralization, refers to the distribution of control, meaning there is no central entity controlling the system, and decision-making is distributed throughout nodes. While all decentralized systems are distributed not all distributed systems are decentralized. This thesis accommodates both: the distributed aspect provides geography scalability, and decentralization provides independent operation and reduced single points of failure.

Moreover, a distributed setup with physical devices enables the devices to sense processes and act on data collaboratively across vast networks [Mar21]. The primary focus of this thesis is on literature relevant to distributed networks with simulated and physical devices. Given its reliance on sensors and actuators, IoT is a subtype of distributed networks [BMZA12]. Consequently, this definition will reference various concepts and key aspects in foundational IoT literature.

Nonetheless, a distributed system does not necessarily consist of physical or simulated devices. Often, distributed networks are defined as a distributed set of processes [CL85; Mar21]. Research and definitions of distributed processes are not discussed further, as they are not directly relevant to this thesis. However, key insights from this literature will still be considered, as it presents comprehensive methodologies, approaches, and fundamental aspects of distributed systems applicable to networks containing physical or simulated devices. Concepts such as Fog Computing, Cloud Computing, or Edge Computing will also be considered. Although these technologies are not the main focus of this work, their underlying principles and approaches are still relevant.

Turning to the distribution aspect, the term 'distributed' in distributed networks highlights the geographical dispersion of system components [CDKB11; LL90]. Thus, components can be spread across large geographical areas, contrasting the centralized nature of traditional cloud systems [BMZA12].

One key aspect of distributed networks is scalability, enabling flexibility and applicability in different scenarios [AIM10; CDKB11; TS07]. Further key aspects include scalability in terms of size and administrative dimensions, which are both of great relevance for this work [TS07; ZZT10]. Size scalability refers to the ability to add more devices to the network, while maintaining a flexible structure that can adapt to a dynamic and potentially fluctuating number of devices. [TS07]. The administrative scalability is equally significant since it allows easy management despite spanning various independent administrative organizations [TS07]. Therefore, the system's ability to scale its resources and efficiently manage larger datasets or workloads is critical to the functionality of a distributed network within the scope of this work.

Another key aspect is interoperability, which is crucial due to the heterogeneous nature of devices within the network [BMZA12; CDKB11; TS07]. Various connected devices, ranging from high-performance mainframes to simple and lightweight sensors, contribute to this heterogeneity [TS07]. Interoperability allows devices to interoperate despite having different Operating Systems (OSs) and standards [BMZA12]. Therefore, it is possible to create a distributed network across domains to provide seamless services, regardless device heterogeneity [BMZA12].

A further essential aspect of distributed networks is the unified, non-distributed view of the network presented to users. Distributed systems should appear as a single system to users since the system behaves as a unified, non-distributed entity. Differences in hardware, OS, or geographical location of components should be hidden from the user.

For instance, data representation differences refer to varying formats across devices, such as different file systems or encoding types, which the system must reconcile automatically to maintain consistent usability. Hiding complexities ensures usability and enhances reliability and user trust in the system [TS07]. These factors are essential for building distributed networks that ensure seamless user experiences and enable reliable communication across heterogeneous devices.

The independence of components in such a network allows each computer to operate entirely autonomously [LL90; TS07; ZZT10]. Furthermore, since processes execute concurrently in such a network, they do not necessarily synchronize with each other except through explicit message passing. Concurrent execution of such processes allows them to run independently without requiring synchronization [CL85]. Due to this asynchronous operation of each component in the network, where no global controlling instance exists, processes operate at their own pace, which creates issues in event ordering and synchronization [Lam19]. While independence enhances scalability and fault tolerance, it also introduces complexity in coordination and consistency across the network.

A centralized approach addresses problems caused by asynchrony of individual devices, utilizing a central controller that handles devices in a distributed network [CL85; DG08; KA19; Lam19; SEL19; TS07]. The central host monitors the network, coordinates communication, and ensures efficient task distribution among devices. This level of centralized control requires maintaining a global state across the system [TS07].

In distributed systems, a global state represents a collective snapshot of all local states across all processes in the system and the communication channels between them at a certain point in time [CL85; Lam19]. This global state forms the foundation for consistent snapshots in distributed networks, enabling better control from a centralized host [Lam19]. Subsequently, such a global controller enables overcoming the geographical space between the infrastructure elements [KA19].

A global controller is crucial for a robust distributed system since distributed networks lack shared memory and synchronized clocks as each process can only record its local state [CDKB11; Lam19]. Thus, balancing the workload of all devices is essential for the seamless integration of such a network. That can be achieved by dynamic task allocation, which ensures that all machines in the cluster are effectively utilized by managing distributed data across the network [AIM10; DG08]. In conclusion, a central controlling host is indispensable for addressing asynchrony, maintaining a global state, and balancing workloads, all of which are critical for scalable and efficient distributed networks [CL85; DG08].

While the inclusion of a centralized host utilized for management enables the user to have insights into the state of the network, this does not necessarily mean the network architecture is centralized since the aforementioned host can be utilized just for management tasks and network administration. Therefore, despite a central administrative host, each node can operate autonomously, making decisions based on local information and interactions with neighboring nodes [ZZT10]. This centralized host can bring significant advantages in addressing asynchrony, maintaining a global state, and distributing tasks to available and suitable nodes [CL85; Lam19; TS07]. For this work, the central host primarily serves as a facilitator, enabling not only the coordination of devices but also the deployment into the distributed network without restricting the autonomy of individual devices [ZSA+14].

The decentralized approach emphasizes distributing control and decision-making across the network [KA19; ZZT10]. The autonomy of each device in a distributed network plays a key role in the decentralization approach. Since each device must be capable of making decisions based on local information and interactions with other devices within the network, there is no direct central unit managing the entire system [ZZT10]. The absence of a fully controlling and managing central host prevents single points of failure. It reduces reliance on a central decision-making system since it can continue functioning even if individual nodes fail [HD21; ZZT10]. Furthermore, the decentralized approach enhances the performance, latency, fault tolerance, and scalability of such a system [TS07; ZSA+14; ZZT10].

While the decentralized aspect of a distributed system plays a critical role in distributing control and decision-making in the network, effective communication between devices is equally essential for ensuring coordination, consistency, and overall system reliability in distributed networks, especially in a decentralized context [ZZT10]. The communication between the devices is realized via message passing, ensuring concurrent execution of processes, asynchronous communication, and fault detection, which are all essential factors of a distributed network [CDKB11; CL85; LL90]. Ensuring reliable and ordered message delivery is critical to the system design [Lam19].

With network bandwidth being an essential part of communication within the network, it poses a significant risk of being a bottleneck [DG08]. Therefore, it is vital to consider network bandwidth as a limited resource when configuring or developing a distributed network.

The aforementioned communication can, for example, be established using the Message Queue Telemetry Transport (MQTT) protocol, ensuring the integration of all essential aspects required for robust communication in a distributed, decentralized network. MQTT is a lightweight, publish-subscribe messaging protocol designed for minimal bandwidth and low-power devices. It is not only widely utilized, particularly in the context of IoT, but also serves as a robust and secure communication protocol for distributed environments composed of heterogeneous devices.

To maintain seamless communication and functionality, distributed systems in real-time applications must minimize latency, as the geographical distribution of devices increases the risk of latency issues [BMZA12; CDKB11]. Following the discussion on potential issues and bottlenecks in distributed networks, it is crucial to understand how to manage failure in order to guarantee flawless operation despite latency issues or bandwidth limitations.

Since a system can continue operating correctly despite failures of some components or network disruptions, mechanisms should be in place to recover from failure, handling it accordingly and ensuring the reliability of the system [DG08; TS07]. Nonetheless, the heterogeneous nature of devices increases the risk of partial failures, which can potentially compromise the entire system and must therefore be effectively managed [DG08; Lam19; ZZT10].

As detecting device failure in a centralized way presents a significant challenge due to the independence of each device, it is necessary to use approaches that aim to monitor the status of the devices from an external perspective, for example, from a central host managing the network [DG08]. Error handling presents a significant challenge in distributed networks due to the heterogeneous nature of their components, making failure management a fundamental aspect of ensuring robustness [CDKB11; KWND05].

2.2.2 Deployment in Distributed Environments

Software deployment is defined as the set of processes that occur between the acquisition and execution of software, encompassing activities such as release, installation, activation, configuration, and maintenance [Dea07]. Deployment in a distributed and decentralized environment refers to the process of installing, configuring, portioning, managing, and distributing application components across a heterogeneous network [Dea07; KBL+19; KWND05]. This includes configuring and deploying these software components and activating them in a specific order adhering to predefined dependencies, to ensure a seamless deployment process [ESB+14].

A significant factor in enabling the decentralized nature of distributed deployment is the use of a deployment manager that can dynamically determine the placement of objects based on application requirements, user proximity, or network conditions [ZSA+14]. This approach is crucial for this work as it ensures seamless deployment across diverse devices while providing users with a comprehensive overview of the entire orchestration process.

Nonetheless, a significant amount of research discusses a decentralized approach without any central deployment management tool [ESB+14; KBL+19; ZSA+14]. Since utilizing a central deployment management tool is crucial for ensuring the overall state of the systems, this approach is of limited relevance to this thesis. Still, these approaches must be considered, as this work involves significant decentralization within its distributed network.

Formalized description scripts must be utilized to ensure a robust and seamless deployment in a decentralized and distributed environment [CASG13; Dea07; DKMD04; HCJL15; KBL+19; KLM+19; LVCD13].

A common approach for deployment in decentralized and distributed environments involves the creation of a deployment description, which defines the stepwise process of deployment [DKMD04; LVCD13]. Such descriptions define the application's structure, including hosts and their connections [DKMD04]. Additionally, protocols, standards as well as metadata have to be included since they ensure the compatibility between heterogeneous devices, which is crucial for the ability to make deployment possible in a distributed network [HCJL15; LVCD13]. Ultimately, such deployment descriptions are fundamental for this work since they not only provide the proper distribution of software across the network but also define the specific order of deployment based on dependencies and network structure [ESB+14].

An essential factor in creating compelling deployment descriptions is the comprehensive understanding of the topology and geographical distribution of devices within the distributed network since that significantly impacts the functionality and interoperability of included devices [DKMD04; KWND05]. Such a geographical distribution in a decentralized and distributed environment can be modeled with dependency graphs. The deployment order can be deduced via topological sorting, ensuring the proper dependencies [KWND05]. Furthermore, this includes mapping physical devices to platform-specific configurations, thereby reducing manual efforts and errors [HJL15]. To summarize, the documentation of the network topology is crucial for the implementation of an automated deployment framework in a heterogeneous, distributed, and decentralized environment.

Commonly known as semantic service descriptions, deployment descriptions provide a unified semantic representation for heterogeneous things across platforms to address platform interoperability challenges [HCJL15]. This ensures robust provisioning processes by translating platform-specific configurations into semantic metadata using a unified knowledge scheme [HJL15]. A significant amount of literature regarding deployment in distributed environments suggests utilizing formats like JavaScript Object Notation (JSON) to ensure a widely recognized template that is easily interpretable by both tools and humans.

Architectural specifications are dynamic, constraint-based models that manage deployment complexities, making their utilization highly beneficial [Dea07; DKMD04]. These software component specifications include information about the host on which the component is to be executed and the connections between the hosts [DKMD04]. Overall, deployment descriptions based on an architectural approach build an excellent foundation for seamless and secure deployment in distributed environments.

A deployment manager is needed, which not only initiates the deployment process but also incorporates the functionality of creating the descriptions to achieve fully distributed and decentralized deployment with deployment descriptions. [CASG13; ESB+14]. Overall, a central deployment management component is needed to ensure a robust, seamless, and secure deployment of software components in a distributed network of devices.

Although utilizing deployment descriptions is a reliable and robust approach, it is important to note that some literature also explores deployment without relying on specific descriptions. [KBL+19; TGBG20]. Deployment without description documents is of limited relevance to this thesis, as we will incorporate deployment specifications, however, it remains important to acknowledge.

In this context, it is crucial to highlight that the dynamic selection of devices for deployment is fundamental to the flexibility of an automatic deployment framework. Since components and connections can be updated or moved to new hosts dynamically, flexible mechanisms that allow the topology of deployed applications to be rearranged without disrupting other parts are crucial for the functionality of distributed deployment [DKMD04]. Therefore, deployment managers have to decide dynamically on the placement of objects based on application requirements, user proximity, and network conditions. This approach supports the decentralized control and execution of deployment since such flexible mechanisms allow the topology of deployed applications to be rearranged without disrupting other parts [DKMD04; ZSA+14].

Another key aspect is the standardization and modularization of components for deployment to remain consistent within the deployment process [Dea07; DKMD04]. A scalable and modular approach is necessary since deployment processes can benefit significantly from reusable components and standardized modeling. Modularity ensures the reusability of created deployment scripts, so each deployment system can act independently or collaboratively, making it suitable for large-scale deployments in a distributed environment [KBL+19; LVCD13]. Furthermore, this component-based deployment approach, which focuses on modularity, manages configurations and dependencies by packaging needed metadata from the network to ensure all fundamental information is available for the deployment process [Dea07]. In conclusion, having standardized and modular components for distributed deployment is crucial for ensuring consistency within the deployment process.

While the foundational concepts of deployment in distributed networks provide a solid basis, it is equally important to address the key challenges associated with implementing such deployment processes. Above all stands the heterogeneity factor of a distributed and decentralized network, which poses a significant challenge for a seamless deployment process due to the inherently diverse architecture [CASG13; HJL15; LVCD13; TJLG17]. These differences in data structures, representation formats, and communication across devices in different environments create significant challenges in deploying and configuring in a distributed and decentralized network of devices [FRC+13; HJL15; LNP+18; TJLG17]. Furthermore, since distributed services span across multiple ownerships, locations, and network domains, the deployment process and communication configuration become significantly more complicated [LNP+18]. Overall, this lack of standardization leads to considerable issues since seamless deployment is a pivotal aspect of this thesis [FRC+13; HJL15].

Additionally, deployment in a distributed environment entails yet another limitation: the network capacity limits. Deployment failures happen due to limited network capacity which refers not only to the network's actual bandwidth but also to constraints such as poor network conditions, which can impact deployment reliability regardless of automation. These unexpected platform conditions include internal server errors, particularly in unstable or non-commercial platforms [HJL15]. On top of that, overcoming firewalls that block inbound communications without compromising security is crucial, as network device restrictions are a widespread issue, particularly in distributed decentralized environments [KBL+19]. Furthermore, devices often have limited computational and storage capacity, leading to further issues that need to be overcome with a distributed deployment framework [KBL+19]. Altogether, these restrictions posed by limitations of devices and networks not only have to be addressed in this work but also require solutions. Only then can a secure and consistently operating deployment framework be established.

Deployment failures caused by network limitations are entirely independent of whether the deployment is automated or manual. Therefore, robust error handling within the deployment framework is fundamental. Error handling is essential and can be implemented as exception handling to manage new failure modes introduced by the distribution aspect to ensure security and functionality in such a network [KWND05]. Such capability-based protection prevents malicious operations within the entire runtime environment [CM23; DKMD04]. Taken together, addressing the security aspect of deployment in distributed and decentralized networks is critical for this work since the distribution aspect introduces new vulnerabilities that must not hinder fully automated deployment in any way.

Given these key challenges of deployment in such an environment, Ansible[1] provides an effective solution, enabling automated, scalable, and consistent configurations across distributed systems. Ansible is an open-source and automation tool by Red Hat, designed for deployment, managing configurations, cloud provisioning, and service orchestration [CBC+20; GCBC21; Nin23; SSR22; YTY18]. Ansible consists of several key components. Connection plugins manage communication with the hosts. The host directory contains the IP addresses of the target systems and specifies details such as the username, password, and port for deployment. Functional plugins provide additional features, including logging and email notifications [MB20; YTY18]. Finally, playbooks, written in YAML Ain't Markup Language (YAML), define task execution sequences, tasks, variables, templates, handlers and roles for automation [MB20; Nin23; SSR22]. Ultimately, Ansible supports software configuration management, service provisioning, real-time applications deployment, orchestration, monitoring, and logging [YTY18].

Besides that Ansible has an agentless structure, which means there is no need to install any agent on target machines since the deployment works solely by pushing configurations over Secure Shell (SSH), enabling a simplified deployment process and reduced maintenance overhead [CBC+20; GCBC21; Nin23; YTY18]. Therefore it has a simple set up process and is easy to manage, which significantly lowers the barrier for adoption compared to alternatives like Puppet, Chef, or SaltStack [MB20; Nin23].

In addition to this, Ansible provides a scalable and flexible orchestration tool, since it not only supports various tools and manages heterogeneity but also incorporates the dynamic changing of devices throughout the network [GCBC21; Nin23; SSR22]. Hence, Ansible proves to be a suitable orchestration tool for distributed environments, as it can effectively accommodate the heterogeneity of various devices within a distributed network and manage the deployment process there.

Furthermore, automating the deployment process is a crucial aspect of this work, and Ansible emerges as a fitting solution for this purpose, since it reduces time and effort required for manual deployment, whilst enabling automatic deployment and orchestration solutions [GCBC21; MB20; Nin23]. Given that Ansible provides real-time feedback and automated workflows, its integration into a CI/CD-pipeline is straightforward, without any complications or extensive manual configuration [MB20; SSR22; YTY18].

Despite the flexibility and scalability of Ansible, it still has its limitations as its performance on very large systems can be limited or involve some complexity in setup [GCBC21; Nin23]. Nonetheless, Ansible offers a highly suitable solution for this work, as it provides a straightforward and user-friendly approach while addressing all necessary challenges. This is particularly relevant given that the focus of this work

[1] Ansible: https://docs.ansible.com/

is on developing a prototype rather than a fully market-ready solution, which needs to accommodate for large and highly dynamic distributed networks of ECUs.

2.2.3 Testing in Distributed Environments

Testing in a distributed environment, for example a distributed network of ECUs, is defined as executing a system, subsystem or component under specific preconditions with specific inputs, to compare actual behavior with expected behavior, which needs to be predefined by a developer or tester [MSEM24; UK99]. Assessing the preconditions of individual components or devices within the network does not imply that each device is tested in isolation. Rather, every defined precondition contributes to a broader scenario that spans multiple components within the network. These components must collectively meet specific conditions for the scenario to be executed [BWB+22]. This process is known as integration testing, a testing phase where modules or components of a distributed or none-distributed network are combined and tested as a group, to ensure proper interaction between the devices [BWB+22; FS22; UK99; WSG98]. Integration testing specifically ensures the interoperability of devices since it focuses on the communication of components in the network. Additionally, testing distributed systems includes conformance testing, which ensures interoperability, reliability, and compliance with standards for the components under test [WSG98].

To conduct integration tests in a distributed environment while ensuring correct interaction between devices, distributed environment patterns must be defined [PLF18a]. These are also known as test case descriptions, which define the steps a complete test scenario must perform. Further, it defines all conditions each device needs to meet for a successful execution of the testing scenario [JNT22]. Test case descriptions must be written in a way that ensures both machine interpretability and human readability, as this ensures a straightforward way of creating test case descriptions without needing extensive programming knowledge [JNT22; KNR09; TDS+13]. Despite that, an entirely natural language-based approach poses a significant challenge for seamless automation, as the accurate interpretation of test cases described in natural language remains insufficient [TDS+13]. Thus, natural language should be used while maintaining a standardized structure that remains comprehensible to machines [JNT22].

To overcome these limitations, Gherkin[2], a Domain Specific Language (DSL), offers a structured approach that ensures both human and machine readability of test case descriptions. Gherkin is a human-readable test case description language primarily used for behavior-driven development. However, it can also be utilized for describing

[2]Gherkin: https://cucumber.io/docs/gherkin/

test cases outside of behavior-driven development, as it provides a well-defined and structured approach to specifying test cases while remaining comprehensible to both machines and humans [VPBB22].

Using keywords like Feature, Scenario, etc., the Gherkin syntax ensures a robust and unambiguous structure despite the inherent interpretability of human language [VPBB22]. To describe specific test cases, Gherkin syntax utilizes a standardized pattern of three keywords: `GIVEN`, `WHEN`, `THEN`. These steps establish a structure defining what must be provided `(GIVEN)` for a test, the conditions that must be met `(WHEN)`, and the expected outcome under those conditions `(THEN)` for the test to pass. A more detailed description of the gherkin syntax and functionality will be discussed in the specific implementation of our framework.

Ultimately, the use of test descriptions ensures the completeness and correctness of a test. Additionally, well-defined test specifications enhance test compatibility, ensuring the independence of test cases from shared resources. This enables test execution even within a distributed, heterogeneous network of various ECUs. Consequently, this enhances maintainability by emphasizing modularity, reusability, analyzability, and modifiability, ensuring overall flexibility in test execution [JNT22].

Building upon this, model-based testing is a widely adopted and state-of-the-art approach in the automotive industry, where it plays a crucial role in ensuring the reliability and efficiency of complex software systems. Moreover test case driven testing enables the automation of the test development and execution process in the context of automotive testing [KNR09].

The automotive industry has developed ISO 26262[3], a well-established and widely adopted test security standard that ensures compliance with functional safety requirements [JNT22; LS13]. To ensure compliance with automotive security standards, it is essential to consider this standard when developing an automated test execution process. Especially while testing critical components the execution has to follow strongly ISO 26262 to ensure comprehensive testing of critical automotive systems [LS13]. Having this in mind such security standards must be incorporated into the test case description. Ultimately, a robust and recognized approach for testing ECUs begins with defining the testing strategy, followed by specifying test cases, executing predefined test cases, and finally recording the test results to enable subsequent evaluation [JTH21; LBF16].

Due to the heterogeneity of devices, tools, programming languages, and standards, testing in a distributed network is a complex process. Ensuring consistency between system models and test cases, as well as managing an increasing number of configurations, is fundamental to both testing in general and automotive testing. Therefore the correctness

[3]ISO 26262:2018: https://www.iso.org/standard/68383.html

of distributed components such as sensors and actuators needs to be established before conducting tests [FS22; LK21; PLF18b; San16; WSG98]. Automating the entire testing process is crucial for addressing the challenges posed by heterogeneity and its impact on testing [MMBT19; San16]. Automation not only ensures the interoperability of devices within the network itself but also the modularity of the whole network, which enables the developers to dynamically add and remove devices [SFF+24]. Automation facilitates a standardized testing process that effectively addresses heterogeneity issues by eliminating the need for manual intervention, which is inherently error-prone and can result in significant challenges.

2.3 Related Work

This section discusses related work of this thesis, implementing similar approaches and solutions for given issues regarding automation of distributed and decentralized deployment and testing. Related work can be split into two primary research directions: first, distributed and automated deployment and testing in IoT, and second, distributed and automated deployment and testing in the automotive sector. Both directions focus on automation while also largely addressing the distributed nature of such systems.

To address interoperability issues and the high complexity of manual deployment, Hur et al. [HJL15] propose an automated IoT service deployment approach using an Semantic Service Description (SSD) ontology. Their approach aims to standardize IoT service descriptions through a common semantic model and automate service deployment across multiple platforms. By replacing platform-specific configurations with a unified framework, it reduces manual effort and deployment time while improving interoperability between heterogeneous IoT devices and platforms [HJL15]. Similarly, our framework also relies on service descriptions that specify the exact deployment steps within our distributed network of ECUs. Moreover, our approach also aims to address interoperability challenges while reducing manual effort and human error through deployment automation. While the work of Hur et al. [HJL15] focuses solely on automation processes for deploying in distributed environments, this thesis integrates both deployment and testing into a single, seamless workflow.

The SSD ontology by Hur et. al. [HJL15] provides a common framework for describing IoT services, ensuring semantic consistency across platforms. It consists of three main concepts. The first, property, describes static attributes of physical objects, such as name, software version, and hardware specifications. The second, capability, represents data provided by the device, including sensor readings and actuator commands. Lastly, the server profile defines platform-specific interaction details, such as Application Programming Interface (API) endpoints, request formats, and supported actions like deployment, updates, and deletion [HJL15]. The conceptual approach of the SSD ontology shows many similarities to our approach, since our deployment description also includes the aforementioned metadata of devices within the network, used to select dedicated devices for deployment.

Considering this theoretical approach, the proposed workflow starts with the retrieval of semantic metadata (e.g., device configurations) from devices within the networks, using a JSON-LD format to represent the retrieved metadata. Given that, the JSON-LD metadata gets converted into Ressource Description Framework (RDF) models, a data model with well-defined formal semantics based on directed graphs, to then serialize the final platform-specific service description using Apache Jena and JSONLD-Java. Lastly, the proposed framework deploys IoT services to target platforms using given server profiles. With this workflow in mind, our approach also begins with a network scan to retrieve data and configurations from the devices. These are then used to generate dedicated deployment scripts, enabling the deployment of software to suitable devices within the network. Despite having great resemblances in terms of the theoretical approach, the direct implementation of this solution differs greatly from our approach. When it comes to automate deployment in such a distributed network, Hur et. al. suggest the usage of Topology and Orchestration Specification for Cloud Applications (TOSCA), whereas our approach utilizes a more lightweight solution with Ansible, creating less overhead. The only direct implementation similarity lies within the way of storing semantic metadata of the device, which is done in a JSON data format.

Ultimately, the suggested framework provides reduced manual effort, improved deployment speed, and cross-platform compatibility. Despite that, certain platform-specific constraints still require manual adjustments. Furthermore, future works include enhancing reliability especially focusing on platform-specific constraints, optimizing performance, and expanding support for additional IoT platforms and real-world deployment scenarios.

Building on similar concepts, López-Viana et al. [LDP22] propose automating deployment in IoT edge computing by implementing a CI/CD pipeline, using GitOps, a modern approach to managing infrastructure and application deployment using Git as the single source of truth [LDP22]. The objective consists of several key aspects. First, it involves investigating the feasibility of implementing GitOps in IoT Edge Computing. Another important aspect is analyzing the benefits and limitations of GitOps. Additionally, a proof-of-concept will be conducted using Cloud Native Computing Foundation tools to gather experimental findings and insights into the feasibility and limitations of GitOps adoption at the edge. Finally, the study aims to identify challenges in extending GitOps beyond Kubernetes [LDP22].

The proposed workflow consists of three key steps to ensure a seamless deployment. First, the developer pushes code to the GitLab[4] repository, triggering the deployment process. This process includes building, testing, and publishing an application artifact to a Nexus OSS artifact registry. The artifact is then used to generate a deployment descriptor in the form of a Kubernetes YAML manifest, which is subsequently stored in the GitLab repository. Secondly, ArgoCD, a declarative GitOps-based continuous delivery tool for Kubernetes enables automated deployment and synchronization of applications based on Git repositories and monitors the Git repository for changes. Once a new deployment descriptor is detected, ArgoCD automatically deploys the application to the Kubernetes cluster. The edge devices, including Raspberry PIs, then retrieve the latest version of the application and ArgoCD tracks and ensures that the live system state matches the desired state. Lastly, the proposed framework includes automated rollouts, updates, and rollbacks. New versions are deployed seamlessly and, if any issues occur, GitOps can roll back to a previous state automatically [LDP22].

With this approach in mind, notable parallels can be drawn to our work. Our framework also aims to provide automated deployment and test execution on changes in a GitHub repository. It includes tracking of changes on certain software files and then the deployment of these software components to dedicated devices, if changes in those components are detected. Despite significant similarities in the conceptual approach of addressing challenges of manual deployment in a distributed device environment, the direct implementation of the framework differs substantially, as our framework relies on an entirely distinct set of technologies. Nonetheless, our framework also leverages GitHub's direct CI/CD integration to enable change tracking and automated deployment triggers.

Henceforth, the proposed framework enables successful automation of continuous deployment, improved audibility and observability, enhanced disaster recovery and business continuity, and lastly, supporting Git versioning, allowing for precise tracking of infrastructure changes [LDP22].

Nevertheless, the researchers acknowledge certain limitations of this approach. ArgoCD does not manage the physical infrastructure, which means physical or virtual edge nodes must be manually provisioned or automated using external tools like Terraform or Ansible. Furthermore, the usage of these tools lack support for non-Kubernetes environments, which means external tools like Ansible or Terraform may be required to support non-Kubernetes infrastructure. In contrast, our framework directly incorporates Ansible into the automation process of deployment, and therefore, reduces the amount of manual steps. Moreover, the complexity of Kubernetes manifests, requires specialized

[4]GitLab: https://about.gitlab.com/

knowledge to define deployments. Unlike our framework, which uses Gherkin a natural-language based description format, enabling a straightforward creation of deployment descriptions for developers without the need for specialized knowledge.

Ultimately, the work investigated the feasibility of GitOps in IoT Edge Computing by implementing a CI/CD pipeline using Kubernetes, MicroK8s, ArgoCD, and KubeEdge. The study successfully automated deployment but highlighted challenges such as manual infrastructure provisioning, workload constraints, and Kubernetes dependencies.

Additionally, there is related work focusing solely on the automation aspect of deployment without a specific focus on distribution, especially under the use of Ansible as a deployment automation tool. Hereby the scientific work proposes a CI/CD pipeline automation approach using Jenkins, Ansible, and Kubernetes. Consequently, reducing manual intervention and accelerating product iteration through automation of the integration process [MB20; SSR22]. Both works implemented a framework for deployment automation based on the following steps. First, the developer commits and pushes the code to a Git repository. When a developer makes changes to the code, Jenkins detects them and then automatically triggers the deployment pipeline. After this, the deployment itself is handled by Ansible [MB20; SSR22].

In both studies, the proposed framework significantly reduces deployment time compared to manual processes, while also enhancing workflow efficiency and minimizing errors. With regard to the deployment automation, both studies show significant similarities to our work. Our framework also automates deployment through CI/CD integration, utilizing Ansible as deployment tool. The key distinction is that changes are not detected by Jenkins, but rather by a custom GitHub action. Furthermore, our framework offers significantly more than just deployment automation, surpassing the approaches of Mysari and Bejgam [MB20], as well as Singh et al. [SSR22]

It is essential to highlight that all aforementioned related works discussed only the automation of deployment without a specific focus on the automotive industry and its unique challenges. To bridge this gap, several studies have explored automated deployment and testing in the context of ECUs, which play a crucial role in modern vehicle systems.

Novak and Kocourek [NK05] propose an automated test site to evaluate ECU behavior in Controller Area Network (CAN)-networks across, physical, link, and application layers, to reduce the time required for testing, in comparison to traditional manual approaches especially focusing on reducing human intervention throughout the testing process of ECUs [NK05].

The approach consists of automated test procedures, minimizing manual intervention, by comparing ECU behavior against predefined pass and fail criteria. Given that, the process can be divided into four steps. First, the ECU under test is connected to the automated test site and predefined test conditions as supply voltage levels, CAN bus parameters are set and the simulation of the remaining vehicle network is started. Subsequently, the test site automatically runs tests at physical, link, and application layers, measuring voltage levels, bit rate, and signal rise or fall times, evaluating CAN controller behavior and error handling, and ensuring correct frame transmission and ECU response. Following this, the CAN generator injects predefined signals into the network, detects errors and ensures synchronization, while the diagnostic tools read ECU error logs and internal states. Lastly, the software compares test results to the aforementioned predefined limits and the ECU gets marked as `PASS` or `FAIL`, with additional soft- or hardware failure categories [NK05].

To summarize, this study provides a significantly deeper exploration of the automotive domain, with a strong focus on the specific characteristics of ECUs, while still addressing important factors needed to be considered in this work. While the general approach of automatically testing ECUs against predefined test criteria is a fundamental aspect of our framework, the focus lies less on lower-layer communication and more on high-level values and overall ECU behavior. The automation of testing in the context of ECUs significantly reduced test time compared to traditional manual testing, while still providing a precise, automated validation of ECU behavior in CAN networks.

Still, the approach was primarily designed for standalone ECU testing, without incorporating full vehicle systems, which is a crucial factor of our approach, especially focusing on the distribution aspect of the Device Under Test (DUT). Nonetheless, Kanchana and Murthy B.N. [KM20] propose a framework for continuous integration to improve ECU software development through automation of build, deployment, and testing processes using Jenkins. Given that, they aim to reduce manual effort and errors in software integration and flashing, while still enhancing software quality assurance through an automated test framework [KM20].

The proposed framework divides the automated testing and deployment process into three fundamental steps. First, the continuous integration and build automation needs to be executed. Here, software components are stored in a shared Git repository, where developers can commit their code across various locations. Jenkins automatically detects changes within these software components and triggers automated build jobs, seamlessly integrating various DevOps tools such as Maven, Make, and Ant to execute the build process. In case of failure, Jenkins notifies developers with logs and error messages.

Once a successful build is completed, the compiled binary file is automatically flashed onto the ECU, utilizing iSystem Debugger and WinIdea. A Python script is executed to open a pre-configured workspace in WinIdea, download the boot program to the ECU, and initiate the Unified Diagnostic Services (UDS) reprogramming sequence before downloading the main application. Next, the boot loader is executed on the ECU to download the application software and verify the firmware's integrity. After that, the

ECU enters diagnostics mode, to then enable secure flashing utilizing the UDS protocol. After successful verification, the new application software is downloaded and installed [KM20].

The last step of the process is the automated testing using Jenkins and Python scripts. These Python testing scripts are integrated with Jenkins, so they execute automatically after flashing and the test results are logged and saved in an Excel sheet. Furthermore, the UDS services are used for diagnostic tests, ensuring compliance with ISO 14229[5]. The proposed framework utilizes CAN for the communication between ECUs and the tester tool.

Significant similarities can be drawn to our work here, as we also focus on enabling automated deployment and testing for the automotive sector in a distributed environment. It is important to emphasize that our framework will not incorporate the aforementioned automotive standards and ECU communication protocols, as it serves solely as a prototype for demonstration purposes and the theoretical applicability of such a framework in a real-world automotive application. Nonetheless, this work remains highly relevant, as it explores the implementation of such an automation framework while considering automotive standards. Conversely, our work focuses on the distributed aspect of such a framework, which is not addressed by Kanchana and Murthy B.N. [KM20]. In summary, the approach significantly reduces development time, errors, and improves reproducibility and reliability [KM20].

In contrast, existing literature primarily emphasizes the automation of the testing process, with particular attention to system distribution and device heterogeneity. Arthofer et al. [AWS24] propose a test automation framework based on comparing different approaches for test automation in the embedded systems area. They developed a framework called Kubrotest based on a previous analysis of three test automation tools to establish the best state-of-the-art approaches for test automation. Their framework supports acceptance tests, Graphical User Interface (GUI) tests, and general automated regression tests, making it applicable to various scenarios.

First, the proposed framework requires test cases created utilizing the Robot framework analyzed earlier, which are then packaged into separate containers as Kubernetes pods, with each including the test logic in the form of test suites and necessary tools for test case execution. Next, Kubernetes is utilized to distribute the pods across multiple nodes within the cluster, enabling geographically distributed testing.

It is important to know that the distribution of pods does not resemble a complete deployment of software components but solely the test case itself for execution, in contrast to our approach. Subsequently, the test cases get automatically executed

[5]ISO 14229:2020: https://www.iso.org/standard/72439.html

utilizing the Robot framework within each pod, which employs Local Area Network (LAN) to connect to the SICAM 8 devices and subsequently perform actions such as sending ping commands for connection verification or interacting with the device for monitoring and control operations. During execution, the Robot framework logs each test step individually and records each success, failure, and timing accordingly. Lastly, the framework automatically generates Hypertext Markup Language (HTML) reports and logs after each test run, which are then stored in the dedicated database for later evaluation by the developer.

Additionally, Arthofer et al. [AWS24] implemented Azure DevOps pipelines to control when and where tests are triggered and to coordinate the execution of test cases across various test labs, creating an automated and distributed environment for testing and evaluating their framework. Furthermore, the utilization of Kubernetes enables the proposed framework to support decentralized and distributed execution, making it suitable for large-scale testing environments.

Despite the robust approach, including distribution, the solution still lacks addressing the aspect of heterogeneity, which is a highly relevant aspect of our work since the proposed solution solely focuses on SICAM 8 devices explicitly developed for the Siemens Smart Infrastructure EA. Nonetheless, the entire framework poses as a free-of-charge development, built strictly on open-source technologies such as Kubernetes [AWS24].

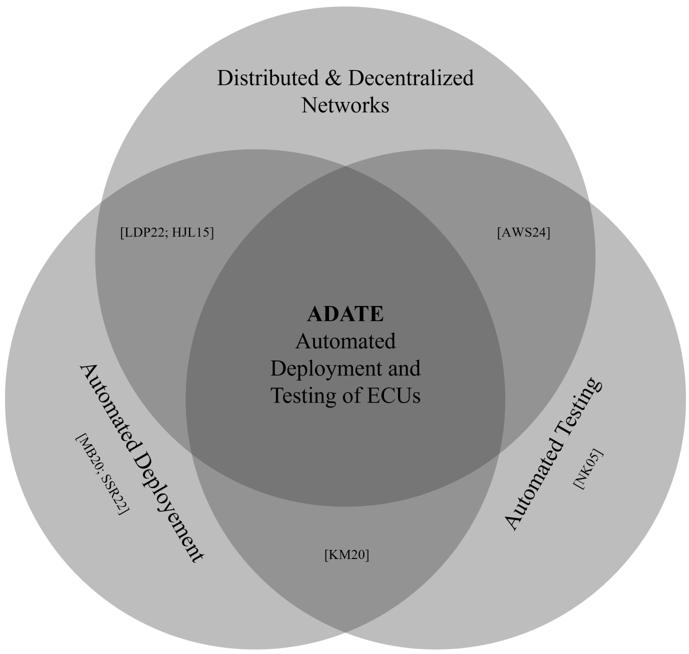

Figure 2.1: Comparative Research Map: Positioning ADATE in the Landscape

In summary, the suggested framework exhibits substantial similarities to our proposed approach concerning testing automation, as we have likewise developed test automation in a distributed environment, including a pipeline that manages the entire automated testing process. However, our solution does, in contrast, include the high heterogeneity and variety of devices posed by real-world use cases of ECUs, not only focusing on one specific device type. Moreover, our framework encompasses testing automation and automating the deployment and undeployment processes of software components onto dedicated devices.

Collectively, a substantial body of research focuses on distributed and decentralized testing or deployment, often integrating various automation approaches. Existing studies often incorporate the aspects of distribution and deployment [HJL15; LDP22], while others focus solely on the automation aspect of deployment [MB20; SSR22]. Additionally, some research integrates automated testing and automated deployment [KM20], whereas others concentrate exclusively on automated testing [NK05]. However, as illustrated in figure 2.1 no existing study combines all three key aspects: distribution, automated deployment, and automated testing into one unified framework. Consequently, our framework ADATE combines all three key components into a single solution, resulting in a seamless and automated process of deployment, testing, and undeployment, thereby addressing the identified research gap.

3 Conceptual Approach

This chapter presents the conceptual approach of our proposed solution, developed in response to the research gaps identified in the previous chapter. The approach enables a high-level architecture for future work addressing automated deployment and testing in distributed environments. Our proposed approach establishes strong parallels relevant to the automotive industry while focusing on a distributed and decentralized development environment of ECUs.

The initial phase of the approach's development involved analyzing research gaps and identifying related works that partially address these issues, and subsequently integrating all critical components into one centralized approach that overcomes the previously discussed research gaps.

Current research encompasses three fundamental fields: (i) distributed and decentralized approaches, (ii) automated deployment processes, and (iii) automated testing processes, most of which are partly interconnected with each other as visualized in Chapter 2, Figure 2.1.

3.1 High-Level Architecture

A comprehensive integration of all three fields has not yet been realized, particularly within the automotive sector. We therefore propose a unified, distributed, and decentralized approach that enables seamless deployment, testing, and undeployment of software components on dedicated ECUs. This approach mitigates the issue of the highly error-prone, manual-driven, and resource-intensive process of current ECU testing. Furthermore, it overcomes the challenges of geographically distributed development while providing developers with the benefits of centralized testing.

This approach would make the currently substantially inefficient development process, which involves iterative shipments of ECUs to and from various development sites, significantly more cost-effective and time-efficient. Therefore the need for physical transfers would be eliminated, as every ECU from the OEM could now be integrated into a globalized network and tested in a geographically distributed manner.

Our approach is based on a decentralized, distributed network enabling OEMs to automate the testing process of ECUs. It consists of two fundamental components: the network, including the devices selected for testing, and the proposed framework, which can be divided into five further components: (i) Backend, (ii) Message Broker, (iii) Deployment Controller, (iv) Testing Controller, (v) Database, as illustrated in Figure 3.1. The backend component coordinates the message broker, deployment controller, and testing controller as shown by the connecting arrows, ensuring data flows in a decentralized yet managed way. All components must reside within the same network, as it is fundamental for the communication between the devices themselves and with the framework.

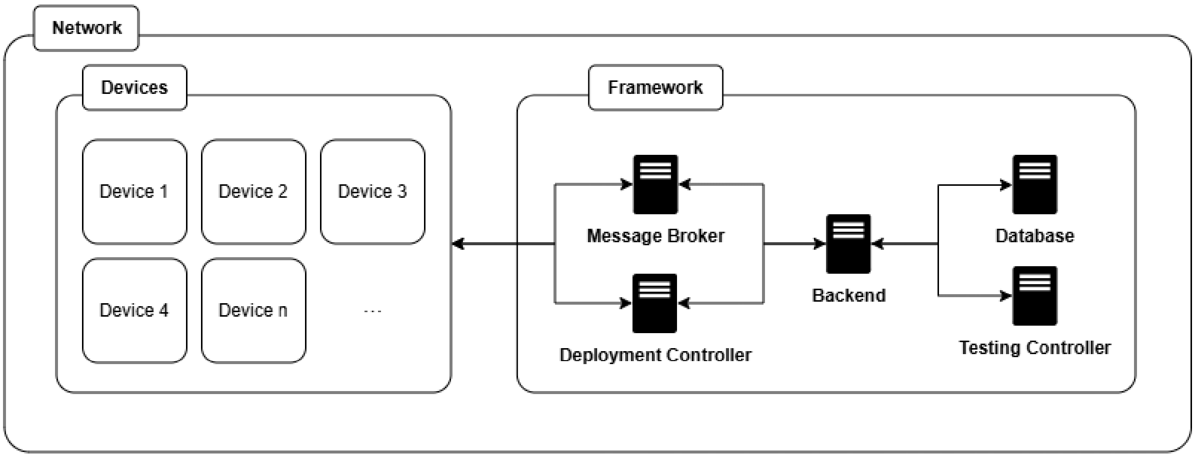

Figure 3.1: High-Level Architecture of the Proposed Conceptual Approach

The cluster of devices within the network contains n simulated or physical devices, enabling testing even in the early stages of development. It is crucial, to ensure scalability of the proposed framework, as it must handle both minimal and extensive numbers of ECUs, including hosting the devices and performing necessary operations associated with them.

(i) Backend: Serves as the central logic unit that orchestrates the interactions between all framework components, managing workflow execution and all data flows.

(ii) Message Broker: Facilitates asynchronous and standardized communication between components, ensuring reliable message delivery [HJL15]. This is crucial as communication is a fundamental key factor that enables the functionality of such a distrusted testing method.

(iii) Deployment Controller: Responsible for distributing and executing deployment and undeployment tasks on the target devices, using predefined instructions. To ensure reliability, deployment and undeployment operations must incorporate robust failure and error handling, as errors and failures within the network can still occur despite well-defined communication [NK05]. This includes a focus on automotive regulations and standards such as ISO 26262.

(iv) Testing Controller: Manages the execution of predefined test scenarios on the deployed software components and collects their results. This includes monitoring the test results and enabling feedback for iterative development of the DUT.

(v) Database: Stores relevant data such as deployment metadata, test outcomes, and device-specific information for further analysis.

3.2 Proposed Workflow

To achieve an efficient and distributed testing process of ECUs all mandatory steps that are currently performed primarily manually, must be automated. These steps include the testing environment setup and incorporation of the ECUs under test, creating dedicated software images and flashing the ECUs with them, conducting the tests, and, finally, evaluating the results [BD10; HJL15; MB20].

The proposed approach leads to the state-of-the-art development process of iterative enhancements and changes of software components while including full automation and removing the necessity for any manual intervention throughout the testing process [BNG+99; YBB06]. Accordingly, the proposed approach must incorporate automated device configuration within the aforementioned distributed setup of ECUs, automated orchestration of software components, and automated testing based on the newly added software components and provided test and deployment specifications by the developer.

We propose six fundamental steps to achieve this: (i) creating a test description, (ii) network scan for devices, (iii) automated creation of deployment specifications, (iv) automated deployment, (v) automated testing, and (vi) automated undeployment, as depicted in Figure 3.2, with the arrow representing the fully automated transition between workflow steps, triggered by changes detected in the Git repository. Thus, forming an iterative development process that enables the reusability of all devices within the network and adhering to state-of-the-art development approaches, while including full automation [BNG+99; YBB06].

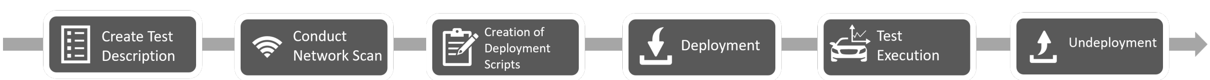

Figure 3.2: ADATE Workflow

Specific implementation details will follow in the subsequent Chapter 4, as this overview serves solely to ensure clear understanding of the frameworks general structure and execution flow.

The test description must be provided by a developer responsible for the testing of devices including both required device specifications and a test scenario description. This ensures a comprehensive workflow description that enables the seamless execution of the described test scenario within the proposed framework. Furthermore, the description forms the structural basis for every subsequent step of our proposed solution, as it is used to deploy, test, and undeploy software components.

The proposed approach must incorporate Continuous Integration (CI) utilizing a version control system such as Git[1], enabling automated builds and tests upon code merges within a dedicated pipeline [LDP22; SSR22]. It not only enables pipeline functionality but also serves as a storage solution for software components intended for testing.

Upon finished configuration, the developer commits the created test description onto the version control system, for the pipeline to initiate the automated deployment process. Once triggered, the pipeline performs the following steps:

1. Automated Deployment And Testing of ECUs (ADATE) conducts a network scan within the network and retrieves the necessary data from all reachable devices to calculate connection strength, which is used for device selection. The established set of devices allows our approach to deploy software components onto dedicated devices to enable subsequent testing.

2. ADATE automatically generates deployment and undeployment scripts based on the developer's provided test case description and software components. Furthermore, it includes all suitable devices for the deployment, selected by predefined criteria.

3. ADATE deploys all software dependencies and components onto the dedicated devices, ensuring the code is executed upon deployment completion. Throughout the deployment process, a robust rollback strategy must be implemented to address potential issues that could otherwise lead to failed deployments. Due to the unpredictable nature of networks, this step is fundamental to ensure flawless deployment.

4. ADATE initiates distributed test execution adhering strongly to the functionality described within the description file. Consequently, the testing framework returns the test result values to the CI/CD pipeline for later evaluation by the developer or tester.

[1] Git: https://git-scm.com/

5. ADATE undeploys each previously deployed software component from all devices, ensuring repeatability of this process.

This automated process simplifies software development and testing for ECUs. As a result, developers can avoid the rigorous and error-prone manual process of setting up deployment and testing environments to validate newly added components through automation.

3.3 Requirements

Due to the diversity and heterogeneity of ECUs, the proposed framework must support incremental changes and modifications to enable stakeholders to adapt more flexibly throughout the development process. A modular architecture is fundamental to ensure the system's adaptability to various ECU types and development scenarios, as it decouples single modules, while functioning together as one system. In summary, our adaptable approach enables its utilization in various scenarios and use cases.

To ensure a well-structured and secure environment, the OEM should also define the permitted and prohibited actions within the distributed network, thereby clarifying the roles and privileges of each participating stakeholder. Thus the OEM retains complete control of the system. At the same time, ECU suppliers are limited to adding or removing their ECUs from the network. They are not granted any privileged access, ensuring overall system security and preventing abuse by external parties. Clear guidelines must be established for suppliers on how to add or remove ECUs, including precise data and configuration requirements. Additionally, the approach must incorporate a security layer implementing various permission rights, ensuring no stakeholder misuses the framework, as each participant is granted permissions strictly aligned with their designated tasks.

Ultimately, the proposed approach combines distributed deployment, automated testing, and modular system design into a unified framework specifically designed to target the challenges of ECU testing. This conceptual foundation guides the practical implementation detailed in the next chapter.

4 Implementation

Building upon the conceptual framework outlined in Chapter 3, this chapter demonstrates its practical realization through an implementation we name ADATE which strictly adheres to the architecture described in Chapter 3. Although the functional prototype is simplified and does not incorporate standardized automotive frameworks or software architectures to avoid unnecessary complexity, it remains sufficient to capture real-world dynamics. Despite the absence of these standards, the framework provides a solid foundation for advancing distributed deployment and testing systems in future developments.

Throughout this chapter, we refer to the simulated devices as ECUs. Note that for our proof-of-concept implementation, we used Ubuntu Linux devices instead of real-world ECUs to decrease complexity and, thus, ease development. However, the term is used for conceptual consistency and will be applied interchangeably.

4.1 Architecture

Our framework, ADATE, consists of three key components: (i) the deployment framework, which also handles undeployment, (ii) the testing framework, and (iii) the network, including the devices, as shown in Figure 4.1. Together, these components form the foundation of an entirely automated and containerized deployment and testing workflow.

Our framework is fully containerized using Docker version 4.36.0[1]. Considering this, containerization provides maximum flexibility, ensuring a completely independent framework setup of the environment on the host machine [HD21; TS07]. Therefore, ADATE can be deployed and executed on any device that supports Docker and has sufficient computational resources to run the framework. This makes our framework not only easy to integrate into existing structures but also highly adaptable for a wide range of real-world use cases.

[1] Docker: https://www.docker.com/

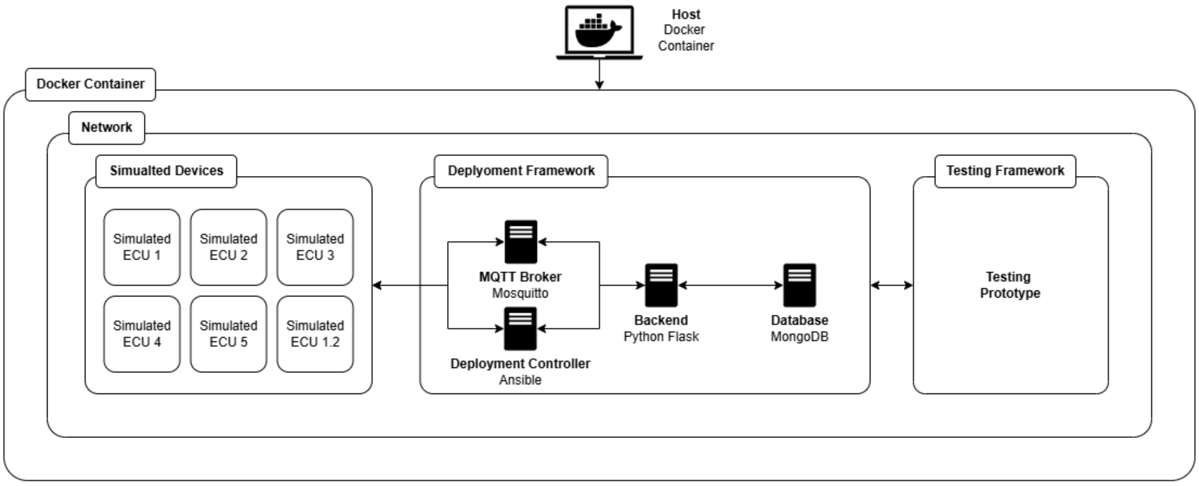

Figure 4.1: System Architecture of the ADATE Prototype

As mentioned, our network consists of diverse devices with various sensors and actuators, representing simplified versions of ECUs, ensuring applicability in real-world scenarios where numerous devices operate with different configurations and software components. To implement this, we developed an entirely simulated set of devices. In particular, these simulated devices function as simple ECUs, designed to enable tasks such as reading or simulating given sensor data. As a direct result of simulating devices, the deployment and testing of the network can take place in an entirely virtualized environment, providing flexibility and cost-efficiency compared to the utilization of physical hardware [CM23].

For this thesis, we utilized Ubuntu 20.04[2]. The simulated ECUs are equipped with various packages to guarantee fundamental functionality. Firstly, basic Linux packages ensuring the core functionality of the simulated device with `bash`, `curl`, `sudo`, and `vim`. Next, we included the Python and Python-pip packages for seamless Python script execution. The tools `net-tools` and `iputils-ping` were then added to enable network configuration and diagnostics, enabling fundamental networking features such as reading the Internet Protocol (IP)-address or pinging other devices within the network that are relevant for debugging and the reliability of the device. Lastly, the OpenSSH-server package was added, providing an SSH server that allows remote access to the Linux system and secure, encrypted connection for remote administration. Collectively, these packages ensure a realistic simulation of a physical device.

ADATE's foundation includes six devices: `ecu_1`, `ecu_1.2`, `ecu_2`, `ecu_3`, `ecu_4`, `ecu_5`, representing five key ECUs needed for the Child Detection Protection Scenario (CDPS) used to showcase the functionality of our framework. With `ecu_1.2` being an exact

[2]Ubuntu Linux: https://ubuntu.com/

copy of ecu_1 to showcase the functionality of the automated device selection based on connectivity strength, which will be elaborated in Section 4.3. Furthermore, these devices exist only for the duration of the Docker container running the framework and are fully reset and erased upon each restart or rebuild. This ensures a clean start, free from residual cached files or potential data corruption.

To enable wireless communication between devices and to ensure functionality in a distributed environment, they are interconnected through a central network using Ethernet over Wireless Local Area Network (WLAN) in contrast to state-of-the-art local communication between ECUs, e.g., using CAN, FlexRay, or Automotive Ethernet [NHB05; NSSW05; Pra19].

Ensuring interoperability and communication across all devices within this WLAN network, our approach implements communication utilizing MQTT. Since MQTT poses a lightweight, easy-to-use, and straightforward implementation of wireless communication, it is the most suitable communication protocol for this network. Although MQTT is not an automotive communication standard, it is well suited for this prototype context due to its lightweight overhead and ease of integration with containerized systems.

Devices within the network must be uniquely identifiable. To achieve this, each newly connected device is assigned a static IPv4 address, which serves as the primary addressing point for communication with the device. Expanding on this, the static IPv4 addresses for the base devices can be found in Table 4.1. Moreover, IPv4 is a widely recognized standard for deployment tools like Ansible. Furthermore, port mapping for external access to the simulated devices was added mapping `localhost:222x` onto `192.168.1.10x:22`, as illustrated in Table 4.1.

Device Name	IPv4 Address	External Access	Software Component	Functionality
ecu_1	192.168.0.101	localhost:2221	door_ecu.py	Simulate Doors
ecu_2	192.168.0.102	localhost:2222	ac_ecu.py	Simulate Air Conditioning
ecu_3	192.168.0.103	localhost:2223	interior_sensor_ecu.py	Simulate Child Detection Sensors
ecu_4	192.168.0.104	localhost:2224	headlight_ecu.py	Simulate Headlights
ecu_5	192.168.0.105	localhost:2225	infotainment_ecu.py	Simulate Infotainment

Table 4.1: Table of Simulated Devices Including IP Address and Core Functionality

To ensure a reliable and comprehensive backend, this work utilizes Python Flask[3], a framework commonly used in various significant software projects. Furthermore,

[3]Python Flask: https://flask.palletsprojects.com/en/stable/

Flask enables an easy-to-use and straightforward way of development, with minimal architectural constraints. As this work constitutes solely as a prototype and not a fully developed industry-ready framework, such a simple backend constitutes an optimal way of incorporating all needed interfaces and functionalities.

Additionally, a robust and seamless working backend requires a database containing all the framework's crucial data. Since no further complex data structure is needed, the database is best implemented straightforwardly, without any unnecessary overhead. Therefore, MongoDB[4] is utilized as it handles JSON objects without complex data structures. To ensure complete functionality and seamless integration into the existing framework, the MongoDB container is connected to the designated Docker network and configured with a port mapping to port 27017, allowing external access to the simulated environment. The MongoDB database includes three foundational collections: `ecu_network`, `id_mapping`, `gherkin`.

The `ecu_network` collection stores all registered devices within the network with all necessary information regarding each device. Furthermore, the database includes all test case descriptions created by the user to ensure the reusability of those in later stages of development. Likewise, the database includes a structured way of saving deployment and test execution jobs in a dedicated `id_mapping` table. Figure 4.2 depicts the complete data structure and the entity relation.

To initialize the database with essential information like registered and available devices, a JSON file populates it at framework startup. This file effectively simulates a developer's registration process when adding new devices to the network, ensuring that all necessary device data is available upon launch.

[4]MongoDB: https://www.mongodb.com/

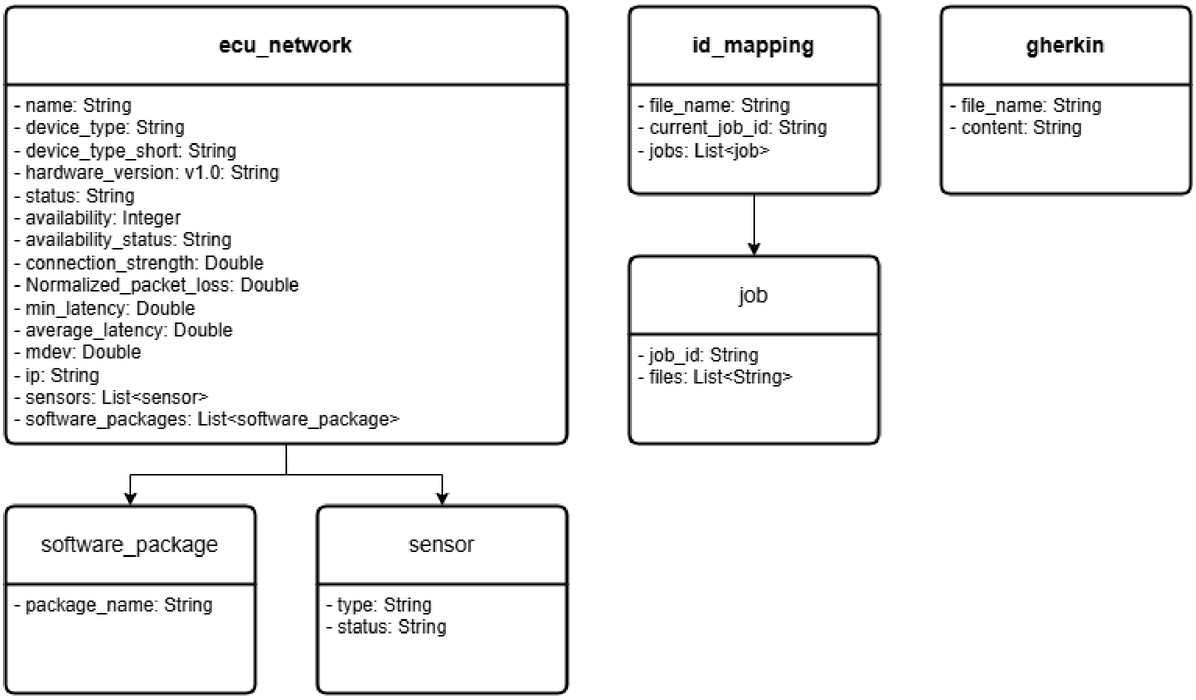

Figure 4.2: MongoDB Data Structure Diagram

With this foundation in place, we can focus on the functionality of the necessary testing and deployment frameworks to demonstrate the feasibility of achieving fully automated test execution through deployment and CI/CD pipeline integration.

4.2 Test Description

Based on the discussion in Section 2.2, Gherkin was selected as the test description language due to its natural-language readability and well-defined syntax combination. It allows test cases to be specified, easily understood by non-technical stakeholders, and is straightforward to implement for developers [SV18].

4.2.1 Structure of Description File

The incorporated test description file is structured as a `.feature` file and adheres strictly to Gherkin syntax. It consists of two main sections: a background section, containing device configuration, data and a scenario section, describing the test process. Each scenario is composed of multiple tests, with each representing a distinct condition within

Listing 4.1 Test Description Gherkin Structure

```
Scenario: Test car is locked
   Given <time_constraint>
   When <sensor1> is <condition1>
   Then <post_test_processing>
```

a defined time frame. These tests, in turn, consist of individual test steps that evaluate specific sensors or actuators against predefined values. A test step is successful only if the actual values match those defined in the test description.

In this context, it is important to clarify the terminology used throughout this work. While Gherkin syntax defines "Features" as the top-level element in a `.feature` file, we refer to each Feature as a Scenario, as it encompasses a complete test description and device data. Likewise, individual "Scenarios" within a Gherkin file are referred to here as "Tests" since each represents a distinct test instance in our framework. This precise terminology is critical for understanding the structure and execution of test scenarios throughout the system.

The test description must include metadata crucial for correct and reliable device selection. Since our setup consists of various heterogeneous devices, the test description must specify which devices are relevant to the scenario described.

Our test description utilizes Gherkin's table syntax to list all required devices and their configurations. For our use case, a single, centralized test description file is the most effective solution, as it ensures consistency, enhances readability, and simplifies maintenance across all scenarios. Splitting the test into multiple files would introduce unnecessary complexity, especially when individual parts are linked to the corresponding device configurations. The exemplary CDPS devices used in our case study are shown in Table 4.1, which directly corresponds to the entries in the Gherkin feature file. While our approach diverges from Gherkin's intended use of tables, it remains syntactically valid. It allows all necessary configuration data to be captured in a single, self-contained file, making it a practical choice for our setup.

The table format for the test description was refined iteratively, balancing readability and completeness to ensure that all device configurations are clearly and comprehensively represented.

After incorporating all the required metadata for a successful deployment into the test description, the next step is detailing the test scenario. The domain-specific language Gherkin utilizes the Given-When-Then structure for test descriptions, where `Given` specifies the initial conditions, `When` defines the action or event, and `Then` describes the expected outcome, as depicted in Listing 4.1.

Listing 4.2 Test Description Example: Scenario: Test car is locked

```
Scenario: Test car is locked
   Given 3 sec passed
   When door_fl is EQUAL to 0
   And door_fr is EQUAL to 0
   And door_br is EQUAL to 0
   And door_bl is EQUAL to 0
   Then, the test should pass
```

Since the test steps of each Gherkin scenario are bound to specific time constraints, the `Given` steps always include a precise time frame within which the condition must be met for the test to succeed, e.g., `Given 3 sec passed`. This is particularly important because the CDPS is a highly time-sensitive scenario. Moreover, many automotive testing use cases are inherently time-dependent, underscoring the importance of our framework's ability to support such time-sensitive evaluations.

Following the `Given` step is the `When` statement, which defines the specific conditions that must be satisfied within the provided time frame. This step identifies which sensor or actuator on each relevant device must be addressed. It does not matter which device is used as long as it meets the necessary hardware and software specifications. To evaluate multiple sensors simultaneously, our framework uses the `And` keyword to define compound conditions, for example: `When <sensor1> is <condition1> And <sensor2> is <condition2>`. Take the example "Test car is locked" from our CDPS. This scenario must assess whether all car doors are closed for the test to pass. Consequently, the expected condition, a value of 0 (closed), is specified individually for each door, as illustrated in Listing 4.2.

Lastly, the `Then` step specifies that the test passes successfully if the aforementioned conditions are met, e.g., `Then the test should pass`.

Since our framework is designed modularly, adding further information to the `Then` steps is possible and does not negatively affect the execution of deployment or testing in subsequent steps. Additionally, such information may benefit other developers reading or working with the test case descriptions.

4.2.2 Child Detection Protection Case Study

To demonstrate ADATE's practical applicability, we implemented it in the context of the CDPS, a safety function strongly recommended by the European New Car Assessment Program (Euro NCAP) and highly relevant for real-world deployment and testing. Our example CDPS use case incorporates the sensors and actuators listed in Table 4.1, reflecting their role within the scenario. The workflow is based on the recommendations from

Euro NCAP and the patented methods described in EP1 669 251 A (EU) and US7,170,401 B1 (US), which together form a solid foundation for our prototype implementation.

The CDPS comprises nine individual tests as depicted in Figure 4.3, each defining specific conditions that need to be met in a certain predefined time frame for successful execution of the CDPS.

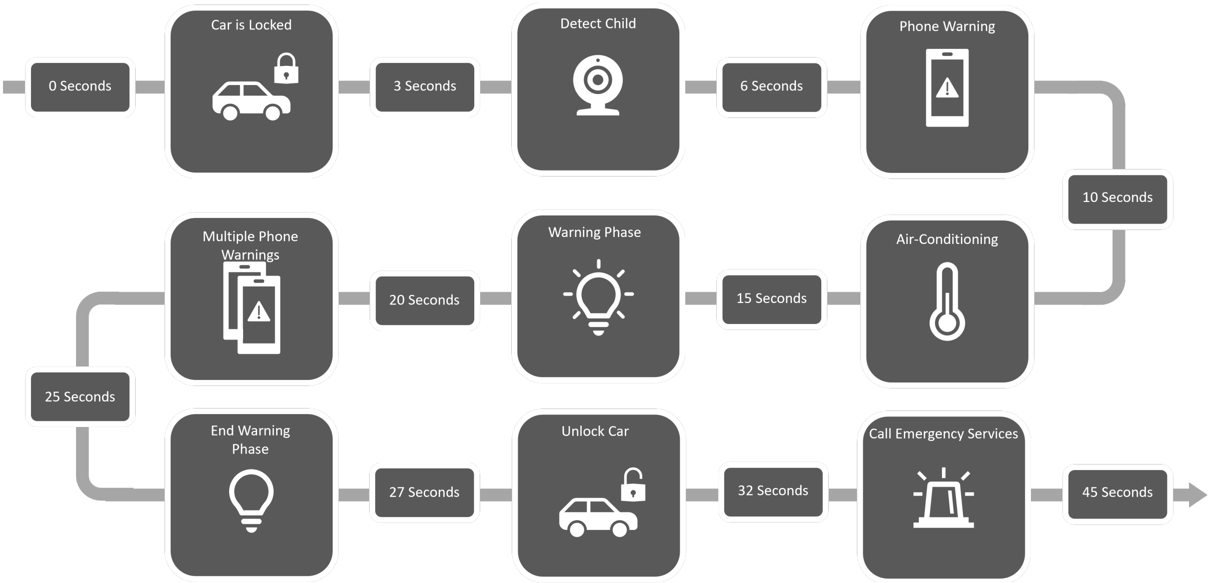

Figure 4.3: Child Detection Protection Scenario Testing Process

First, the scenario begins with `Test car is locked`, during which each door is individually assessed for its status, ensuring that all doors are closed. Furthermore, the car's lock status must return `locked` for the test to pass successfully.

Next, the provided Interior Sensor Management ECU must detect a child within the specified time frame for the car to start the driver warning phase successfully. Detecting a child includes multiple sensors to ensure reliable and redundant detection. The sensors involved are an interior camera, interior radar sensor, microphone, and seat-pressure sensor, all of which must return positive feedback for the test `Detect Child` to pass.

Upon detection of an unattended child, the assessment of the escalation sequence begins with the evaluation of the initial phone warning sent to the driver's device. We assume that the driver has installed the corresponding application for the vehicle, as this is essential for enabling the car to send push notifications to the smartphone. The initial phone warning test `Test Initial Phone Warning` is evaluated against the specific push notification: `notifying driver:  child was detected`.

Next, the `Intervention 1 Air Conditioning` testing process verifies if the climate control started cooling or heating the car depending on the outside temperature, ensuring the well-being of the child. For the test to pass successfully, our test procedure checks whether the air conditioning systems returns the value `23`, as this represents the anticipated value in our fictitious scenario.

Subsequently, the `Test Escalation Warning Phase` monitors the status of the headlights, as the CDPS is designed to warn nearby pedestrians with flashing lights.

After this, another warning message is sent to the phones of all registered car users. For simplicity, this test `Test Intervention 2 Phone Call` is simulated by only assessing one MQTT message `calling all registered car users`. The headlights are monitored again before the headlight warning phase is terminated, meaning the correct workflow would have turned off the headlights at a given moment.

Next, the car unlocks all doors to allow pedestrians to safely retrieve the child from the car. Lastly, the `Test Final Intervention Calling Emergency Services` checks if the car notifies emergency services automatically as a last instance if no reaction from the driver is detected.

4.3 Network Profiler

To ensure reliable deployment execution and to enable the subsequent testing process, information on available and suitable devices must be collected prior to deploying or testing any software components using our framework. As discussed previously, these required devices are specified in the test description.

To discover all available devices, the framework initiates a network scan. However, since network scanning is significantly more complex than the preceding steps and the GitHub[5] Action runner does not have direct access to the internal device network, the pipeline does not perform this operation directly. Instead, it invokes the `network-scan` endpoint of our framework, passing the required `iterations` parameter which specifies the number of iterations the framework should perform.

Given that, the pipeline is implemented utilizing a local GitHub runner hosted on a Windows machine, which is handling the actual scanning process. The backend call utilizes the PowerShell `curl.exe` command to invoke the scanning logic, which in turn utilizes the tools `nmap` and `ping`.

First, `Network Mapper (Nmap)`, an open-source network discovery and security auditing tool used for network inventory, managing service upgrade schedules, and monitoring

[5] GitHub: https://github.com/

host or service uptime, is executed [Das; MAA20]. It is implemented utilizing the `-sn` ("Scan No Port") option, performing only host discovery without scanning any specific ports, as they are irrelevant to our framework. Once the `nmap` scan is completed, all available IP addresses of online devices are first listed and then mapped to the corresponding devices in the MongoDB database. If any of the registered devices' IP addresses are not listed after the successful Nmap scan, the device is flagged as offline and cannot be used for subsequent deployment or testing. Since the exemplary CDPS includes only simulated devices, all relevant devices are listed as online in our use case and are suitable for deployment and testing in later steps.

Next, the `ping` command, a network utility tool used to test the reachability of a host on an IP network and to measure the Round-Trip Time (RTT) for messages sent from the source to the destination, is used, targeting all previously established available IP addresses, which are correspondingly listed in the MongoDB. [WTS03]. Using `ping` enables our framework to gather detailed network data securely [CM23]. The previously mentioned `iterations` parameter specifies the total number of reruns for the `ping` execution. For each target, our framework records the RTT, which enables it to calculate the minimal RTT, maximal RTT, and the average RTT across all `ping` iterations.

In summary, the aforementioned commands enable the framework to efficiently scan large-scale networks to detect live hosts and assess network responsiveness, primarily focusing on the RTT.

Variable	Description
$S_{\text{connection}}$	Connection Strength
L_{avg}	Average Round-Trip Time
$L_{\text{max\_threshold}}$	Max RTT Threshold
$m_{\text{dev\_threshold}}$	Mean Deviation Threshold
m_{dev}	Mean deviation (variability)
P_{loss}	Packet Loss

Table 4.2: List of Symbols Used in the Test Evaluation Formula

Ultimately, the network scans enable us to gather the following metrics for each device: `connection_strength`, `normalized_packet_loss` (Reaching from 0 to 1 or 0% to 100%), `min_rtt`, `max_rtt`, `average_rtt`, `mdev` (the RTT variation). Once these details are obtained for each online device, a single, central metric called `connection_strength` is calculated by combining all relevant network measurements. First, the parameters depicted in Table 4.2 must be derived from previous measurements of each device.

The formula can be structured into three main components: The first is the normalized average RTT:

$$L = \left(1 - \frac{L_{avg}}{L_{max\_threshold}}\right)$$

It is calculated by first defining a maximum estimated threshold for average RTT, which is set to 100 based on the observation that RTT measurements rarely exceed this value. Furthermore, the value is normalized, as the final result ranges from 0 (indicating poor performance) to 1 (indicating excellent performance). The second is the normalized mean deviation:

$$m = \left(1 - \frac{m_{dev}}{m_{dev\_threshold}}\right)$$

This follows the same approach as the normalized average RTT. The threshold is set to 50, based on previous network scans that established typical deviation values. The third is the normalized packet loss:

$$P = \left(1 - \frac{P_{loss}}{100}\right)$$

Since packet loss is expressed as a percentage (0–100%), it is divided by 100 to normalize the value between 0 and 1. The result is then inverted so that higher values indicate better performance and lower values worse performance. The final connection strength is computed by multiplying all three components, ensuring each factor influences the result and significantly reduces the overall value if any component is below 1.

The complete connection strength function is wrapped in both a minimum and maximum operator to ensure values ranging from 0 to 1. This is particularly relevant, as values above 1 could occur if any of the measured metrics fall below their expected threshold, given that both deviation and RTT have no strict upper bound. Taken together, all these ensure one central connection strength value ranging from 0 to 1, resulting in the following function:

$$S_{\text{connection}} = \max\left(0, \min\left(1, \left(L \cdot m \cdot P\right)\right)\right)$$

These calculations and measurements are streamed to the requesting client as an Hypertext Transfer Protocol (HTTP) stream. The GitHub pipeline provides real-time feedback for each ping, allowing the developer to observe the connection strength measurement process directly from the GitHub Action.

Based on the calculated and measured connection and network values, a complete list of devices and their properties will be utilized in the subsequent deployment, testing, and undeployment steps. The above-mentioned list is created by mapping the previously mentioned values to the listed devices in the MongoDB database according to their corresponding IP addresses.

Listing 4.3 Vehicle Access ECU MongoDB Entry after Network Scan


```
{
   "availability": 100,
   "availability_status": "available",
   "average_rtt": 0.06,
   "connection_strength": 0.9989402759999999,
   "device_type": "heating_ventilation_and_airconditioning_ecu",
   "device_type_short": "HVAC",
   "hardware_version": "v1.0",
   "ip": "192.168.1.102",
   "max_rtt": 0.084,
   "mdev": 0.023,
   "min_rtt": 0.037,
   "name": "ecu_2",
   "normalized_packet_loss": 0.0,
   "sensors": [
      {
         "status": "available",
         "type": "temperature"
      },
      {
         "status": "available",
         "type": "status"
      }
   ],
   "software_packages": [],
   "status": "online"
},
```


As a result, the database now holds a comprehensive entry for each device, including its online status and detailed network metrics used to calculate connection strength. An example of such an entry for the VAE (Vehicle Access ECU) is shown in Listing 4.3. This subprocess enables the framework to identify and prioritize devices with optimal connectivity when generating deployment scripts.

4.4 Change Monitoring

Once the developer commits and pushes the modified code to the GitHub repository, the GitHub Action first detects which code files are new or changed before the framework can start any further process. It is important to note that we will be using the terms pipeline and GitHub Actions interchangeably, as GitHub Actions serve as the framework for defining and managing CI/CD pipelines within GitHub. Consequently, the terms are closely related in the context of this work.

The change monitoring mechanism determines which test descriptions are affected by the latest commit and therefore need to be considered for the subsequent deployment, testing, and undeployment. This workflow has two fundamental components: First, the change detection is directly within the test description. Second, changes to the software components awaiting deployment. Ultimately, both changing the test description and adding a new one, as well as editing or adding new software components for the devices, initiates the entire deployment process.

The pipeline begins by listing all modified files, which are identified as those with changes between the latest commit and its predecessor. It then iterates over the list of directories, filtering specifically for the `src/test_description/` path, which contains all test description files in the framework. If changes are detected in specific software components instead, the pipeline executes an additional backend call to the `get-feature-file` endpoint to retrieve a list of test description files corresponding to the modified software component. This step enables automated recurring testing of entire scenarios by modifying individual software components within the scenario.

After identifying all relevant description files that can trigger the automated deployment and testing process, our framework extracts the filenames for later usage. If no relevant changed files can be found, the pipeline returns an error message.

Finally, the changed files selected get transformed into a temporary JSON file, which is used to create the so-called `matrix` of the pipeline. This matrix includes a JSON list of all needed test description names, enabling sequential execution of deployment, testing, and undeployment of each test description individually.

4.5 Deployment

Following the identification of modified test descriptions, the ADATE framework proceeds with generating and executing deployment scripts for the affected software components. Similar to the network scan, executing the entire deployment process within the pipeline would be overly complex. Therefore, the pipeline delegates this task to the backend by calling the `create-deployment` endpoint, which initiates the deployment script creation.

The creation of these scripts is required for the subsequent deployment of software components, as they define the concrete instructions and execution order the process must follow.

For the deployment to be instantiated, the framework requires the test description file name as a string, since it contains information on deployment details, including device configurations and corresponding packages and software components. Considering this, the pipeline retrieves the identified test description file name as previously explained and forwards it to the backend endpoint as a string.

First, ADATE creates a unique job ID, which is then utilized by all subsequent stages to determine the association between deployment files and their corresponding deployments and testing processes. Specifically, our framework implements the Universally Unique Identifier (UUID) 4 python package, a module that generates a random UUIDs with a high certainty of uniqueness, which is crucial to guarantee distinctiveness among created job IDs.

ADATE accesses the test description `.feature` file described in Section 4.2 and extracts fundamental deployment information from the dependency table, which contains information on software components and their respective target devices.

First, the framework must map each job ID to its corresponding `.feature` file, since `.feature` files can be utilized for multiple deployment and testing processes. More specifically, each `.feature` file exists as a distinct data object within the collection, containing a list of all associated job IDs and a current job ID to track the actively running job as depicted in Section 4.1, Figure 4.2.

Thereafter, the framework must create several distinct scripts that handle deployment, test execution, and undeployment. ADATE generates a suitable test case description for the external testing framework incorporated into our work, which we primarily treat as a black box since its detailed functionality lies beyond the scope of this work. The current version of the test framework requires a dedicated JSON file with predefined test steps, devices, and logic blocks. Our framework implements a parser that converts the test description from `.feature` Gherkin syntax into the custom-written JSON format required by the existing testing framework.

Upon completion, the deployment and undeployment scripts need to be created. As already mentioned, our framework utilizes Ansible for deployment and undeployment tasks, which requires an inventory file containing IP addresses of all devices on which deployment is to be performed.

Building on this, ADATE creates the playbooks used for deployment and undeployment. Ansible uses these playbooks to orchestrate the software components onto the devices within the distributed network. Each device receives a dedicated deployment and undeployment playbook, following the naming convention: `<device_name>_<job_id>`. This naming convention is crucial, as each playbook is stored in the same directory titled `<job_id>`. Furthermore, it enables scalability for our framework and allows for parallel deployment and undeployment. After successfully generating all deployment, undeployment, and testing scripts, the backend returns the dedicated job ID, allowing the developer or pipeline to initiate the subsequent deployment process based on the provided job ID.

The framework incorporates a specific API endpoint accessed via `/deploy`, for both the pipeline to call the deployment or the developer to access it directly for debugging purposes. Furthermore, the endpoint requires a list of devices targeted for deployment to ensure that only needed deployment scripts are executed.

The deployment process itself consists of an iterative execution of the `ansible-playbook <playbook_name> -i <inventory_path> -vv` command, specifying the playbook, which gets executed and using the previously created inventory file to map the playbook to the right device and IP address. Moreover, the `-vv` parameter is used to access more output throughout the Ansible deployment process as again calling this endpoint returns an HTTP stream for real-time deployment progress from within the GitHub pipeline itself, ensuring straightforward debugging for the developer.

To illustrate this deployment process, our framework uses the aforementioned simulated devices in Table 4.1 to simulate the CDPS. Thus, each device has dedicated software components deployed onto it, as depicted in Table 4.1 in columns of software component and functionality.

These software components enable the simulated devices to act as if they were ECUs required for the CDPS, with the necessary sensors and actuators. Furthermore, they enable the simulation of the entire CDPS and test its functionality with ADATE. In this case, these components are Python scripts simulating various sensor values, representing the otherwise highly complex software images that need to be flashed onto the ECUs. This poses a straightforward solution while still providing a good demonstration of automation in the process of deployment.

In a real-world automotive scenario, the framework would not just deploy Python packages and scripts but build a complete image specifically designed for the designated ECU. Such a solution could be easily integrated into our framework as it is designed to be completely modular with separate docker containers for each functionality, making switching out the deployment creation process itself straightforward. For example, the integration of Yocto-based build systems, custom flashing routines, or Over-the-Air (OTA) update pipelines could be implemented within the same modular structure.

4.6 Testing

The subsequent testing process is implemented analogously to the preceding steps, as the pipeline extracts the created job ID to map the current deployment execution to the dedicated testing process.

After a successful deployment, the GitHub pipeline proceeds by calling the ADATE API endpoint `/test`, which initiates the testing process of the external framework and returns an HTTP response stream for real-time progress updates. This design choice ensures the modularity of our framework and maintains its independence from external components, making it highly adaptable. Replacing the testing framework with an alternative is therefore straightforward. Nonetheless, we will still concisely address the core functionality of the testing framework as it is part of the entire workflow and incorporates the execution of the CDPS.

In previous work, we developed a fully functioning testing framework, not only specialized on the CDPS use case but also incorporating the distribution aspect and managing the heterogeneity of various devices throughout the testing process. This testing framework enables communication over MQTT between the devices and is, therefore, perfectly suitable as the testing engine of ADATE. As developing an entirely novel testing framework would exceed the scope of this work, we incorporated the already developed and functioning testing framework specifically designed for this use case.

The testing backend consists of three fundamental components: the Flask application, MySQL[6] database, and a Redis[7] database.

The JSON file must be passed to the testing backend containing all fundamental details on the dedicated scenario to initiate a specific testing scenario within the distributed setup of heterogeneous devices. Our framework parses the `.feature` test description into the required JSON file for the testing framework, which is more abstract while containing the same information as the `.feature` file.

Following that, the scenario is described within the JSON file similar to the previously discussed `.feature` file ADATE uses for deployment. A representative example of this is the first step of the CDPS, assessing the `closed` status of all car doors, as depicted in Listing 4.5. The specific test steps are not redefined, as they have already been specified in the preceding `sensor` section of the JSON file. Nonetheless, they remain linked to the dedicated test by their corresponding ID. It is important to note that these IDs do not correspond to the IDs in the MongoDB, as their purpose is solely to bind test step

[6]MySQL: https://www.mysql.com/
[7]Redis: https://redis.io/

Listing 4.4 Device: "ecu_1_door_fl" – JSON File for Testing Framework

```
"sensors": [
  {
    "id": 1,
    "ecu_name": "ecu_1",
    "mqtt_path": "ecu_1/door_fl",
    "key": "door_fl",
    "name": "ecu_1_door_fl",
    "teststeps": [
      {
        "id": 1,
        "operator": "EQUAL",
        "expected_value": "0",
        "name": "door_fl",
        "description": null,
        "post_action": null
      },
      {
        "id": 18,
        "operator": "EQUAL",
        "expected_value": "1",
        "name": "door_fl",
        "description": null,
        "post_action": null
      }
    ]
  },
  ...
]
```

with test within the JSON file. Additionally, the test description can contain logic blocks specifying a time constraint.

Lastly, the JSON file includes the core definition of logic blocks, including `IF` and `TIME` constraints, to ensure correctness. An illustrative example in this study is again the JSON test file created explicitly for the CDPS, which is partly depicted in Listings 4.4 and 4.5 to demonstrate the structure of the document.

Once all preprocessing steps are completed, the test execution begins by retrieving the scenario ID and calling the testing framework backend via: `/scenarios/<scenario_id>/start`. This call returns multiple status Uniform Resource Locators (URLs), each representing a test composed of several test steps.

The returned URLs correspond to each test within the CDPS, including `Test car is locked`, `Detect Child`, `Test Initial Phone Warning`, `Test Intervention 1 Air Conditioning`, `Test Escalation Warning Phase`, `Test Intervention 2 Phone`

Listing 4.5 Test: "`Test car is locked`" – JSON File for Testing Framework

```
"tests": [
  {
    "id": 1,
    "name": "Test car is locked",
    "teststeps": [
      1,
      2,
      3,
      4
    ],
    "logic_blocks": [
      {
        "id": 1,
        "logic_type": "TIME_CONSTRAINT",
        "condition_test_id": 1,
        "time_constraint": 3,
        "next_test_true": null,
        "next_test_false": null,
        "description": "Time constraint: 3 seconds passed"
      }
    ]
  },
  ...
]
```

`Call`, `Test End of Escalation Warning Phase`, `Test Intervention 3 Open Doors`, and `Test Final Intervention Calling Emergency Services`. Together, these tests form the complete CDPS and can be executed in parallel by the framework, provided there are no defined test interdependencies, or manually by the developer. This approach ensures maximum testing efficiency.

ADATE iteratively calls each passed status URL and streams the response as an HTTP stream, continuing this process until the entire test scenario is concluded, with each test resulting in either a pass or a fail. The resulting states of any given test are consolidated into one central JSON test summary file, which is stored for later inspection and debugging within the pipeline environment. It describes all results in detail for each test step, as depicted for an exemplary successfully passed test in Listing 4.6.

Listing 4.6 "Test Car is Locked" – Successful Test Result JSON


```
{
   "test_id": "1",
   "status": "completed",
   "result": {
      "status": "passed",
      "teststeps": [
         {
            "door_fl": {
               "mqtt_value_expected": "0",
               "mqtt_value_passed": "0",
               "status": "passed"
            }
         },
         {
            "door_fr": {
               "mqtt_value_expected": "0",
               "mqtt_value_passed": "0",
               "status": "passed"
            }
         },
         {
            "door_bl": {
               "mqtt_value_expected": "0",
               "mqtt_value_passed": "0",
               "status": "passed"
            }
         },
         {
            "door_br": {
               "mqtt_value_expected": "0",
               "mqtt_value_passed": "0",
               "status": "passed"
            }
         }
      ]
   }
},
```

4.7 Undeployment

The entire workflow concludes with our framework ADATE undeploying all software components of each device after a successful test execution. It is important to note that a "successful test execution" does not imply that all tests have passed but rather that the execution process was completed successfully, yielding either positive or negative results.

The undeployment process follows an inverse approach to the previously discussed deployment, maintaining structural consistency using Ansible playbooks. Consequently, the process commences by terminating the Python processes running on the simulated ECU, followed by removing all Python packages, and concludes with uninstalling all system packages, leaving a clean Linux system ready for further use. Hence, the entire usability of the framework and existing devices for other scenarios and developers can be ensured after the execution of the pipeline.

Collectively, the framework provides a seamless and efficient approach for automatically testing developed code within a distributed environment of simulated ECUs. It eliminates the need for software engineers to manually configure or set up devices, which reduces errors and increases cost-effectiveness. Developers must only provide a test description for the code to be tested. The automation pipeline and ADATE generate and execute the necessary deployment, undeployment, and testing scripts. This approach simplifies the development process while ensuring reliable verification of code functionality.

5 Evaluation and Discussion

Having discussed our theoretical approach and its implementation utilizing various state-of-the-art approaches and technologies, it is crucial to evaluate our framework to ensure its applicability in a real-world use case. We analyzed literature comparing automated, semi-automated, or manual approaches in deployment and testing, specifically focusing on explicitly defined KPIs. As detailed in Section 2.1, we primarily focused on literature, including the distribution aspect. Additionally, the identified literature includes complete test automation, which refers solely to software-based test generation, unlike our approach, which relies on test descriptions provided by software engineers. Despite this, the KPIs suggested by the researchers will still be considered, as they focus on evaluating test automation, which is highly relevant to our work.

The majority of literature focuses on evaluating the automation process of testing, with a lack of research evaluating the automation of deployment and none exploring the automation of undeployment. Nonetheless, we incorporated a limited body of literature regarding the evaluation of deployment and undeployment automation, as they discussed highly relevant KPIs.

Literature comparing automated and manual approaches addresses the following key areas. Based on these key areas, we prioritized the deployment automation evaluation KPIs and the testing automation evaluation, as these incorporate similar approaches and technologies and find direct applicability in our field.

- Evaluation of automated against manual testing, focusing on test automation based on a provided test description, similar to our approach [AWS24; DMQ19; FNL+19; HKS21; KEK22; MLS+23; MZA18; TDS+13; WS14].
- Automation of automated against manual testing, focusing on regression testing, with automation based on provided software and automated dedicated test case generation [BCR08; ESČP17; SGP+19; SM20; VBO+16].
- Evaluation of automated deployment against manual approaches, which is of high relevance as our framework combines both testing and deployment [CM24; LS17].

We evaluated explicitly mentioned KPIs in the identified literature separately for each distinct field. Moreover, we combined our findings into four KPIs with the highest relevance for our work. Predominantly, our research focused on measurable quantitative KPIs as these enable a fast and comprehensive evaluation process of both manual and automated approaches, allowing for a mathematical comparison. The following four KPIs were selected for our suggested evaluation: (i) `Time`, (ii) `Fault Detection`, (iii) `Cost`, (iv) `Manual Intervention`, as depicted in Table 5.1 with a detailed description and the corresponding count of studies and their dedicated citations.

KPI	Detailed Description	Occurrences	Citations
Time	How much time does it take the developer to go through the entire process.	10	[AWS24; BCR08; CM24; DMQ19; HKS21; KEK22; LS17; MLS+23; MZA18; WS14]
Fault Detection	How much errors does the approach detect.	8	[BCR08; DMQ19; ESČP17; FNL+19; KEK22; MLS+23; SGP+19; TDS+13]
Cost	How much resources does it cost to go through the entire process.	6	[AWS24; KEK22; MLS+23; MZA18; VBO+16], suggested by: ([HKS21])
Manual Intervention	How often does the developer need to intervene in the automated process	1	[TDS+13]

Table 5.1: Set of Selected KPIs for Future Evaluation of the Framework

This set of KPIs enables us to evaluate our framework against manual or semi-automated approaches with consistent and quantifiable performance indicators. Key indicators central to this evaluation are the `Time` and `Cost` KPI, as these serve as an indicator of system efficiency, showing which approach is less resource-intensive and more time-efficient. Furthermore, the selected KPIs include `Fault Detection`, which highlights differences between approaches in the number of detected code defects, thereby reflecting the quality of the tests. Lastly, we included `Manual Intervention`, a measurement that counts the number of manual actions a developer must perform within a single testing cycle, indicating the degree of automation enabled by the framework. It is important to note, this KPI is mentioned only once in given set of literature, nonetheless we included it as the automation aspect is fundamental for the evaluation.

5.1 Evaluation Strategy

As previously mentioned we suggest the comparison against both a fully manual and a semi-automated testing approach. The term "fully manual" approach here describes, the process of manual deployment of required software components, establishing connections between devices, initiating the test scenario, specifying each required test case, and finally, undeploying all software components and removing any dependencies from the devices.

A "semi-automated" approach, is structured in contrast to the "fully manual" approach, with a set of automated components or individual core tests, while the remaining software components and tests endure manual intervention. We suggest the utilization of our testing framework as the automated component, as this enables partial automation, making the testing and result generation process automated, while leaving device configuration and deployment of software components manual. In summary, the suggested evaluation is divided into three key components: (i) fully automated testing using ADATE, (ii) semi-automated testing utilizing our testing framework, (iii) fully manual testing.

Each of these components must be conveyed by the same developer to ensure comparability between the approaches. We suggest including developers with varying degrees of testing experience ranging from minimal experience to maximal experience, as this will strongly impact the results of each approach. Additionally we propose conducting each test iteration concluding all approaches N times to reduce variability and ensure the results are generalizable. For each developer all KPIs depicted in Table 5.1 need to be recorded. After N iterations per developer, the average value for each KPI is calculated using the following formula.

$$KPI(x,y)_{avg} = \frac{1}{N}\sum_{i}^{N} KPI(x,y)_i$$

- x: Represents a specific KPI
- y: Represents a specific Approach (manual, semi-automated, automated)
- $KPI(x,y)_{avg}$: Average result for KPI x with approach y
- $KPI(x,y)_i$: Result for KPI x in run i with approach y
- N: Number of iterations taken for each developer

Upon completion of the test execution by each developer all measured KPIs get calculated for all three approaches accordingly, making it possible to compare the efficiency of each approach for each developer. This process is depicted in Figure 5.1. Ultimately, this evaluation process would enable us to create a robust comparison between current state-of-the-art manual and semi-automated testing approaches in a distributed environment against our novel fully automated approach.

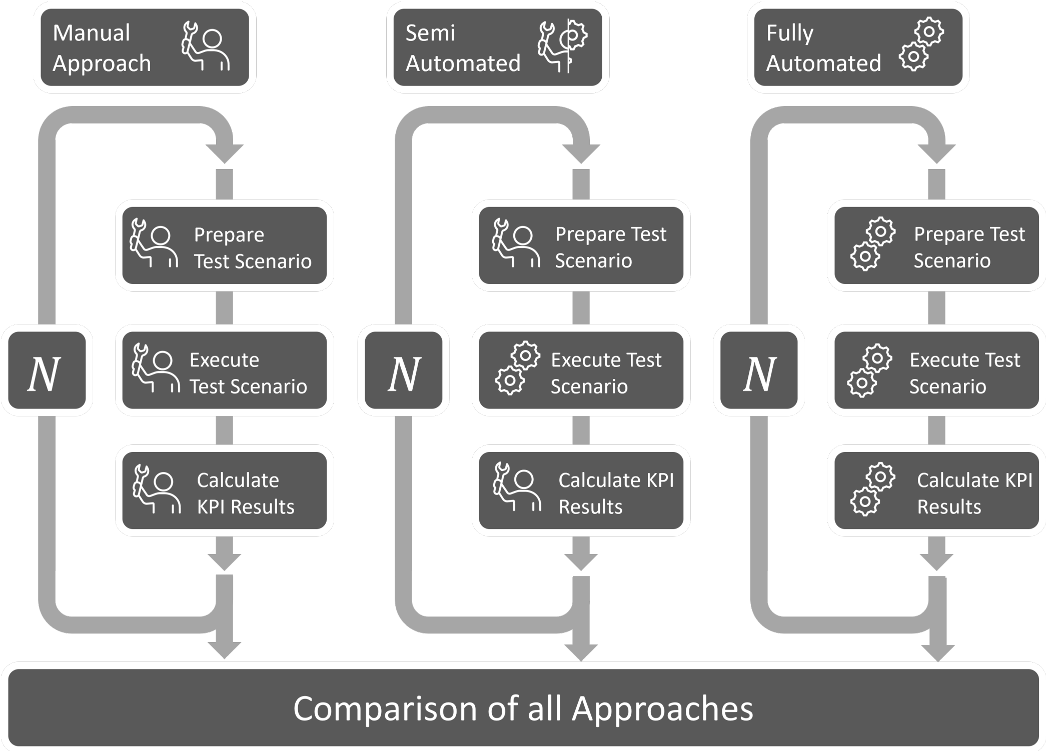

Figure 5.1: Evaluation Workflow for each Developer

While the development and conceptual approach of this framework was focusing primarily on the automotive industry and testing of ECUs, it is nonetheless applicable for various distributed environments, as its core concepts are not domain-specific. Key components such as CI/CD-driven deployment using Ansible, and the distributed test execution logic, are designed modular and do not rely on ECU-specific logic. This makes the overall approach transferable to other distributed environments specifically including Linux-based nodes, making our implementation flexible and reusable in various scenarios.

5.2 Limitations

Our implementation relies on the CI/CD pipeline implemented as a GitHub Action, thereby binding our framework to GitHub and making its application in different software versioning systems more challenging, particularly when migrating to other pipelines such as Jenkins or GitLab CI. Additionally, the CI/CD pipeline automation is currently implemented using self-hosted runners, specifically developed for Windows machines. Therefore, the implementation relies on Windows PowerShell for all commands, such as calling the backend of our framework, making the utilization of self-hosted runners on Linux-based OSs unsuitable.

Currently, the pipeline executes the deploy-test-undeploy cycle sequentially for all modified or newly added description files or software components, leading to a process of test execution that is not optimally efficient. This oftentimes creates waiting times for developers despite computing resources not being fully utilized. Moreover, the pipeline lacks smart scheduling algorithms, which could schedule tasks that utilize the same components consecutively, saving significant resources, as some can be directly reused without requiring separate deployment.

Another important limitation concerns the scalability of the framework in real-world applications. While the prototype demonstrates effective functionality within a network including a limited amount of simulated ECUs, challenges would arise when scaling to hundreds or thousands of physical devices or ECUs. MongoDB might encounter performance bottlenecks at higher data volumes, specifically under frequent concurrent access, while nonetheless being sufficient for prototyping. Moreover, network bandwidth and latency could become significant factors, potentially affecting deployment speed and test execution in geographically distributed environments. Future research of the framework should therefore consider optimizations for distributed database architectures, fault-tolerant communication protocols, and efficient resource management to ensure both scalability and robustness.

6 Conclusion and Future Works

This thesis first established key requirements for a distributed and decentralized environment, focusing on automating the deployment, testing, and undeployment of all dependencies within this environment to support an iterative development process. To achieve this, a systematic literature review was conducted. Relevant literature was identified and grouped into three fundamental areas: (i) distributed and decentralized networks, (ii) automated deployment, and (iii) automated testing, with emphasis on their intersection.

Our evaluation of distributed and decentralized network literature revealed that such a network typically consists of geographically spread, heterogeneous, and independent devices, enabling scalability, interoperability, and communication between the devices via message passing. Moreover, robust automated and distributed deployment requires a description file and needs to overcome the challenges of heterogeneity in a distributed environment while still ensuring consistency, modularity, and security. Additionally, the automation of testing in distributed environments requires a dedicated test specification, which is often realized using a natural-language-based approach. Ultimately, these key aspects should be taken into account during the development process of a framework for automating distributed deployment and testing, particularly in an automotive context.

Subsequently, we identified a significant research gap, which pertains to the intersection of distributed and decentralized networks, distributed deployment automation, and distributed testing automation, specifically focusing on an automotive context. Current research focuses solely on individual aspects or on combinations involving only a subset of these approaches.

Based on the identified research gap and specified requirements throughout the systematic literature review, we propose a conceptual approach for a framework bridging the current research gap of combining distributed deployment and testing into one automated and coherent testing workflow for ECUs. Furthermore, we developed a complete testing process incorporating state-of-the-art concepts and approaches, ensuring the inclusion of core aspects regarding distributed, decentralized, and automotive environments.

Next, we developed ADATE, a framework implementing our conceptual approach, incorporating all required steps for real-world utilization. We present its applicability in a real-world use case through an exemplary implementation of a complete deploy-test-undeploy cycle using the CDPS, based on a use case defined by Euro NCAP.

ADATE utilizes modern container-based simulation and lightweight messaging protocols to mimic real-world ECU behavior. The framework is subdivided into six fundamental steps: (i) creating a test description, (ii) conducting a network scan, (iii) creating deployment scripts, (iv) deployment, (v) test execution, and (vi) undeployment.

Our framework requires a dedicated test and deployment description specified in a `.feature` file utilizing Gherkin syntax, ensuring a well-defined test case specification and a detailed description of the software components. The test case description must be developed and provided by the software engineer conducting the deploy-test-undeploy cycle.

The workflow of ADATE is integrated seamlessly into a CI/CD pipeline process, implemented using GitHub Actions, ensuring full automation, initialized through Git commits. Once all software components are successfully deployed onto the dedicated devices within the network, our framework, conducts all specified test scenarios with real-time monitoring of results.

Upon completed execution of all test cases, the framework generates a result file for all test cases, specifying all test results, including expected and actual sensor data. Lastly, ADATE concludes the deploy-test-undeploy process with the undeployment of all software components to ensure repeatability of the entire workflow for further test scenarios or its utilization by other developers.

In conclusion, with ADATE, we provide a real-world application of our theoretical approach and a concrete implementation for automation in a distributed environment of ECUs. Moreover, we present with CDPS a comprehensive use case showcasing the potential of our framework for industrial utilization.

Lastly, we concluded our research by defining a state-of-the-art KPIs set to evaluate our framework against traditional manual or semi-automated approaches, assessing its applicability in real-world use cases. Therefore, we conducted another systematic literature review evaluating automated testing and deployment. From this, we extracted the most commonly used KPIs and ranked them based on their frequency of occurrence in the literature and their relevance to our work.

Ultimately, we identified the following four state-of-the-art KPIs: (i) `Time`, measuring the overall time taken to conduct an entire deploy-test-undeploy cycle with our framework and the comparative manual or semi-automated approach. (ii) The `Fault Detection`, counting the number of errors detected for each run, ensuring the quality of testing of all approaches (iii) `Cost`, measuring the overall resource expenditure, including developer effort and required infrastructure. (iv) `Manual intervention`, assessing the number of steps requiring human interaction during the deploy-test-undeploy cycle.

Future work should focus on evaluating the framework in real-world industrial settings using the established KPIs. This would validate its practical applicability as a distributed and automated development approach for the automotive industry while uncovering potential production environment limitations. The current pipeline executes deployment and testing sequentially for all detected changes. Efficiency could be improved by integrating parallelization techniques and enhancing the CI/CD process with smarter scheduling and prioritized device queuing.

Bibliography

[AIM10] L. Atzori, A. Iera, G. Morabito. "The Internet of Things: A survey." In: *Computer Networks* 54.15 (Oct. 2010), pp. 2787–2805. ISSN: 1389-1286. DOI: 10.1016/j.comnet.2010.05.010. URL: https://www.sciencedirect.com/science/article/pii/S1389128610001568 (visited on 01/05/2025) (cit. on pp. 22, 23).

[AWS24] S. Arthofer, S. Wilker, T. Sauter. "Development of a Test Automation Framework based on a Comparison of different Approaches for Test Automation in the Embedded Systems Area." In: *2024 IEEE 7th International Conference on Industrial Cyber-Physical Systems (ICPS)*. ISSN: 2769-3899. May 2024, pp. 1–6. DOI: 10.1109/ICPS59941.2024.10640021. URL: https://ieeexplore.ieee.org/document/10640021 (visited on 03/18/2025) (cit. on pp. 37, 38, 67, 68).

[BCR08] A. Bacchelli, P. Ciancarini, D. Rossi. "On the Effectiveness of Manual and Automatic Unit Test Generation." In: *2008 The Third International Conference on Software Engineering Advances*. Oct. 2008, pp. 252–257. DOI: 10.1109/ICSEA.2008.66. URL: https://ieeexplore.ieee.org/document/4668116 (visited on 03/18/2025) (cit. on pp. 67, 68).

[BD10] S. van der Burg, E. Dolstra. "Automated Deployment of a Heterogeneous Service-Oriented System." In: *2010 36th EUROMICRO Conference on Software Engineering and Advanced Applications*. ISSN: 2376-9505. Sept. 2010, pp. 183–190. DOI: 10.1109/SEAA.2010.10. URL: https://ieeexplore.ieee.org/document/5598096/?arnumber=5598096 (visited on 11/06/2024) (cit. on pp. 15, 16, 43).

[BDA+99] P. Bourque, R. Dupuis, A. Abran, J. Moore, L. Tripp. "The guide to the Software Engineering Body of Knowledge." In: *IEEE Software* 16.6 (Nov. 1999). Conference Name: IEEE Software, pp. 35–44. ISSN: 1937-4194. DOI: 10.1109/52.805471. URL: https://ieeexplore.ieee.org/abstract/document/805471/authors#authors (visited on 01/14/2025) (cit. on p. 19).

[BMZA12] F. Bonomi, R. Milito, J. Zhu, S. Addepalli. "Fog computing and its role in the internet of things." In: *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*. MCC '12. New York, NY, USA: Association for Computing Machinery, Aug. 2012, pp. 13–16. ISBN: 978-1-4503-1519-7. DOI: 10.1145/2342509.2342513. URL: https://dl.acm.org/doi/10.1145/2342509.2342513 (visited on 01/05/2025) (cit. on pp. 22, 25).

[BNG+99] J. Beaumont, A. D. Noble, M. Gebhardt, G. Stier, S. Furry. *Automation of ECU Software Development: From Concept to Production Level Code*. English. SAE Technical Paper 1999-01-1174. ISSN: 0148-7191, 2688-3627. Warrendale, PA: SAE International, Mar. 1999. DOI: 10.4271/1999-01-1174. URL: https://www.sae.org/publications/technical-papers/content/1999-01-1174/ (visited on 03/12/2025) (cit. on p. 43).

[BWB+22] J. Beilharz, P. Wiesner, A. Boockmeyer, L. Pirl, D. Friedenberger, F. Brokhausen, I. Behnke, A. Polze, L. Thamsen. "Continuously Testing Distributed IoT Systems: An Overview of the State of the Art." en. In: *Service-Oriented Computing – ICSOC 2021 Workshops*. Ed. by H. Hacid, M. Aldwairi, M. R. Bouadjenek, M. Petrocchi, N. Faci, F. Outay, A. Beheshti, L. Thamsen, H. Dong. Cham: Springer International Publishing, 2022, pp. 336–350. ISBN: 978-3-031-14135-5. DOI: 10.1007/978-3-031-14135-5_30 (cit. on p. 30).

[CASG13] F. Caglar, K. An, S. Shekhar, A. Gokhale. "Model-driven performance estimation, deployment, and resource management for cloud-hosted services." In: *Proceedings of the 2013 ACM workshop on Domain-specific modeling*. DSM '13. New York, NY, USA: Association for Computing Machinery, Oct. 2013, pp. 21–26. ISBN: 978-1-4503-2600-1. DOI: 10.1145/2541928.2541933. URL: https://dl.acm.org/doi/10.1145/2541928.2541933 (visited on 11/06/2024) (cit. on pp. 26–28).

[CBC+20] A. Cepuc, R. Botez, O. Craciun, I.-A. Ivanciu, V. Dobrota. "Implementation of a Continuous Integration and Deployment Pipeline for Containerized Applications in Amazon Web Services Using Jenkins, Ansible and Kubernetes." In: *2020 19th RoEduNet Conference: Networking in Education and Research (RoEduNet)*. ISSN: 2247-5443. Dec. 2020, pp. 1–6. DOI: 10.1109/RoEduNet51892.2020.9324857. URL: https://ieeexplore.ieee.org/document/9324857/?arnumber=9324857 (visited on 11/19/2024) (cit. on p. 29).

[CDKB11] G. Coulouris, J. Dollimore, T. Kindberg, G. Blair. *Distributed Systems: Concepts and Design*. 5th. USA: Addison-Wesley Publishing Company, Apr. 2011. ISBN: 978-0-13-214301-1 (cit. on pp. 22–25).

[CL85] K. M. Chandy, L. Lamport. “Distributed snapshots: determining global states of distributed systems.” In: *ACM Trans. Comput. Syst.* 3.1 (Feb. 1985), pp. 63–75. ISSN: 0734-2071. DOI: 10.1145/214451.214456. URL: https://dl.acm.org/doi/10.1145/214451.214456 (visited on 01/05/2025) (cit. on pp. 22–24).

[CM23] B. Choi, E. Medina. “Building an Ansible Learning Environment for Network Automation.” en. In: *Introduction to Ansible Network Automation: A Practical Primer*. Ed. by B. Choi, E. Medina. Berkeley, CA: Apress, 2023, pp. 543–592. ISBN: 978-1-4842-9624-0. DOI: 10.1007/978-1-4842-9624-0_11. URL: https://doi.org/10.1007/978-1-4842-9624-0_11 (visited on 11/19/2024) (cit. on pp. 28, 48, 56).

[CM24] P. S. Chatterjee, H. K. Mittal. “Enhancing Operational Efficiency through the Integration of CI/CD and DevOps in Software Deployment.” In: *2024 Sixth International Conference on Computational Intelligence and Communication Technologies (CCICT)*. Apr. 2024, pp. 173–182. DOI: 10.1109/CCICT62777.2024.00038. URL: https://ieeexplore.ieee.org/document/10596596 (visited on 03/24/2025) (cit. on pp. 67, 68).

[Das] M. Dasci. “Exploring NMAP Tool_ A Comprehensive Analysis, 28-1.” en. In: () (cit. on p. 56).

[Dea07] A. Dearle. “Software Deployment, Past, Present and Future.” In: *Future of Software Engineering (FOSE ’07)*. May 2007, pp. 269–284. DOI: 10.1109/FOSE.2007.20. URL: https://ieeexplore.ieee.org/document/4221626/?arnumber=4221626 (visited on 01/07/2025) (cit. on pp. 25–27).

[DG08] J. Dean, S. Ghemawat. “MapReduce: simplified data processing on large clusters.” In: *Commun. ACM* 51.1 (Jan. 2008), pp. 107–113. ISSN: 0001-0782. DOI: 10.1145/1327452.1327492. URL: https://dl.acm.org/doi/10.1145/1327452.1327492 (visited on 01/05/2025) (cit. on pp. 21, 23–25).

[DKMD04] A. Dearle, G. Kirby, A. McCarthy, J. C. Diaz y Carballo. “A Flexible and Secure Deployment Framework for Distributed Applications.” en. In: *Component Deployment*. Ed. by W. Emmerich, A. L. Wolf. Berlin, Heidelberg: Springer, 2004, pp. 219–233. ISBN: 978-3-540-24848-4. DOI: 10.1007/978-3-540-24848-4_15 (cit. on pp. 26–28).

[DMQ19] I. Dobles, A. Martínez, C. Quesada-López. “Comparing the effort and effectiveness of automated and manual tests.” In: *2019 14th Iberian Conference on Information Systems and Technologies (CISTI)*. ISSN: 2166-0727. June 2019, pp. 1–6. DOI: 10.23919/CISTI.2019.8760848. URL: https://ieeexplore.ieee.org/document/8760848 (visited on 03/18/2025) (cit. on pp. 67, 68).

[ESB+14] X. Etchevers, G. Salaün, F. Boyer, T. Coupaye, N. De Palma. "Reliable self-deployment of cloud applications." In: *Proceedings of the 29th Annual ACM Symposium on Applied Computing*. SAC '14. New York, NY, USA: Association for Computing Machinery, Mar. 2014, pp. 1331–1338. ISBN: 978-1-4503-2469-4. DOI: 10.1145/2554850.2554951. URL: https://dl.acm.org/doi/10.1145/2554850.2554951 (visited on 01/15/2025) (cit. on pp. 25–27).

[ESČP17] E. Enoiu, D. Sundmark, A. Čaušević, P. Pettersson. "A Comparative Study of Manual and Automated Testing for Industrial Control Software." In: *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. Mar. 2017, pp. 412–417. DOI: 10.1109/ICST.2017.44. URL: https://ieeexplore.ieee.org/document/7927994 (visited on 03/18/2025) (cit. on pp. 67, 68).

[FNL+19] L. Ferreira, S. Nogueira, L. Lima, L. Fonseca, W. Ferreira. "Initial findings on the evaluation of a model-based testing tool in the test design process." In: *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. ISSN: 1949-3789. Sept. 2019, pp. 1–6. DOI: 10.1109/ESEM.2019.8870140. URL: https://ieeexplore.ieee.org/document/8870140 (visited on 03/18/2025) (cit. on pp. 67, 68).

[FRC+13] N. Ferry, A. Rossini, F. Chauvel, B. Morin, A. Solberg. "Towards Model-Driven Provisioning, Deployment, Monitoring, and Adaptation of Multi-cloud Systems." In: *2013 IEEE Sixth International Conference on Cloud Computing*. ISSN: 2159-6190. June 2013, pp. 887–894. DOI: 10.1109/CLOUD.2013.133. URL: https://ieeexplore.ieee.org/abstract/document/6740239 (visited on 11/06/2024) (cit. on p. 28).

[FS22] J. A. Fadhil, Q. I. Sarhan. "A Survey on Internet of Things (IoT) Testing." In: *2022 International Conference on Computer Science and Software Engineering (CSASE)*. Mar. 2022, pp. 77–83. DOI: 10.1109/CSASE51777.2022.9759705. URL: https://ieeexplore.ieee.org/document/9759705/?arnumber=9759705 (visited on 11/06/2024) (cit. on pp. 30, 32).

[GCBC21] M. Gupta, M. N. Chowdary, S. Bussa, C. K. Chowdary. "Deploying Hadoop Architecture Using Ansible and Terraform." In: *2021 5th International Conference on Information Systems and Computer Networks (ISCON)*. Oct. 2021, pp. 1–6. DOI: 10.1109/ISCON52037.2021.9702299. URL: https://ieeexplore.ieee.org/document/9702299/?arnumber=9702299 (visited on 11/19/2024) (cit. on p. 29).

[HCJL15] K. Hur, S. Chun, X. Jin, K.-H. Lee. "Towards a Semantic Model for Automated Deployment of IoT Services across Platforms." In: *2015 IEEE World Congress on Services*. ISSN: 2378-3818. June 2015, pp. 17–20. DOI:

10.1109/SERVICES.2015.11. URL: https://ieeexplore.ieee.org/abstract/document/7196498 (visited on 11/06/2024) (cit. on p. 26).

[HD21] Q. Huang, M. Dong. "Analysis and Design for the Decentralized Network Sharing of Static Resources." In: *Proceedings of the 2020 4th International Conference on Electronic Information Technology and Computer Engineering*. EITCE '20. New York, NY, USA: Association for Computing Machinery, Feb. 2021, pp. 885–890. ISBN: 978-1-4503-8781-1. DOI: 10.1145/3443467.3443873. URL: https://dl.acm.org/doi/10.1145/3443467.3443873 (visited on 01/15/2025) (cit. on pp. 24, 47).

[HJL15] K. Hur, X. Jin, K.-H. Lee. "Automated deployment of IoT services based on semantic description." In: *2015 IEEE 2nd World Forum on Internet of Things (WF-IoT)*. Dec. 2015, pp. 40–45. DOI: 10.1109/WF-IoT.2015.7389024. URL: https://ieeexplore.ieee.org/document/7389024/?arnumber=7389024 (visited on 11/06/2024) (cit. on pp. 15, 16, 26, 28, 32, 33, 39, 42, 43).

[HKS21] K. R. Halani, Kavita, R. Saxena. "Critical Analysis of Manual Versus Automation Testing." In: *2021 International Conference on Computational Performance Evaluation (ComPE)*. Dec. 2021, pp. 132–135. DOI: 10.1109/ComPE53109.2021.9752388. URL: https://ieeexplore.ieee.org/document/9752388 (visited on 03/18/2025) (cit. on pp. 67, 68).

[JNT22] K. Juhnke, D. Neumüller, M. Tichy. "A Quality Model and Checklists for Reviewing Automotive Test Case Specifications." en. In: *Software Quality: The Next Big Thing in Software Engineering and Quality*. Ed. by D. Mendez, M. Wimmer, D. Winkler, S. Biffl, J. Bergsmann. Cham: Springer International Publishing, 2022, pp. 84–104. ISBN: 978-3-031-04115-0. DOI: 10.1007/978-3-031-04115-0_6 (cit. on pp. 30, 31).

[JTH21] K. Juhnke, M. Tichy, F. Houdek. "Challenges concerning test case specifications in automotive software testing: assessment of frequency and criticality." en. In: *Software Quality Journal* 29.1 (Mar. 2021), pp. 39–100. ISSN: 1573-1367. DOI: 10.1007/s11219-020-09523-0. URL: https://doi.org/10.1007/s11219-020-09523-0 (visited on 01/29/2025) (cit. on p. 31).

[KA19] H. I. Kobo, A. M. Abu-Mahfouz. "A Distributed Control System for Software Defined Wireless Sensor Networks Through Containerisation." In: *2019 International Multidisciplinary Information Technology and Engineering Conference (IMITEC)*. Nov. 2019, pp. 1–6. DOI: 10.1109/IMITEC45504.2019.9015923. URL: https://ieeexplore.ieee.org/document/9015923 (visited on 01/15/2025) (cit. on pp. 23, 24).

[KBL+19] K. Képes, U. Breitenbücher, F. Leymann, K. Saatkamp, B. Weder. "Deployment of Distributed Applications Across Public and Private Networks." In: *2019 IEEE 23rd International Enterprise Distributed Object Computing Conference (EDOC)*. ISSN: 2325-6362. Oct. 2019, pp. 236–242. DOI: 10.1109/EDOC.2019.00036. URL: https://ieeexplore.ieee.org/document/8945012/?arnumber=8945012 (visited on 01/06/2025) (cit. on pp. 25–28).

[KC07] B. Kitchenham, S. Charters. "Guidelines for performing Systematic Literature Reviews in Software Engineering." In: 2 (Jan. 2007) (cit. on p. 19).

[KEK22] T. Kurmaku, E. P. Enoiu, M. Kumrija. "Human-based Test Design versus Automated Test Generation: A Literature Review and Meta-Analysis." In: *Proceedings of the 15th Innovations in Software Engineering Conference*. ISEC '22. New York, NY, USA: Association for Computing Machinery, Feb. 2022, pp. 1–11. ISBN: 978-1-4503-9618-9. DOI: 10.1145/3511430.3511433. URL: https://dl.acm.org/doi/10.1145/3511430.3511433 (visited on 03/18/2025) (cit. on pp. 67, 68).

[KLM+19] A. Krishna, M. Le Pallec, R. Mateescu, L. Noirie, G. Salaün. "Rigorous Design and Deployment of IoT Applications." In: *2019 IEEE/ACM 7th International Conference on Formal Methods in Software Engineering (FormaliSE)*. ISSN: 2575-5099. May 2019, pp. 11–20. DOI: 10.1109/FormaliSE.2019.00011. URL: https://ieeexplore.ieee.org/document/8807489 (visited on 01/15/2025) (cit. on p. 26).

[KM20] A. Kanchana, C. Murthy B.N. "Automated Development and Testing of ECUs in Automotive Industry with Jenkins." In: *2020 IEEE International Conference on Electronics, Computing and Communication Technologies (CONECCT)*. July 2020, pp. 1–5. DOI: 10.1109/CONECCT50063.2020.9198423. URL: https://ieeexplore.ieee.org/abstract/document/9198423?casa_token=qBjR6PDnXjMAAAAA:5jUrE0CYJyGW7JIJfhJ1EFzmwh7G-V7NIVb1rbAcO-nsIwXSnbMHLGSpDrp2WRJN6nvo7nCZ5Q (visited on 01/30/2025) (cit. on pp. 36, 37, 39).

[KNR09] E. H. Kim, J. C. Na, S. M. Ryoo. "Implementing an Effective Test Automation Framework." In: *2009 33rd Annual IEEE International Computer Software and Applications Conference*. Vol. 2. ISSN: 0730-3157. July 2009, pp. 534–538. DOI: 10.1109/COMPSAC.2009.188. URL: https://ieeexplore.ieee.org/document/5254082 (visited on 01/30/2025) (cit. on pp. 30, 31).

[KWND05] G. N. C. Kirby, S. M. Walker, S. J. Norcross, A. Dearle. "A Methodology for Developing and Deploying Distributed Applications." en. In: *Component Deployment*. Ed. by A. Dearle, S. Eisenbach. Berlin, Heidelberg: Springer, 2005, pp. 37–51. ISBN: 978-3-540-32281-8. DOI: 10.1007/11590712_4 (cit. on pp. 25, 26, 28).

[Lam19] L. Lamport. “Time, clocks, and the ordering of events in a distributed system.” In: *Concurrency: the Works of Leslie Lamport*. New York, NY, USA: Association for Computing Machinery, Oct. 2019, pp. 179–196. ISBN: 978-1-4503-7270-1. URL: https://doi.org/10.1145/3335772.3335934 (visited on 01/06/2025) (cit. on pp. 21, 23–25).

[LBF16] G. Lami, I. Biscoglio, F. Falcini. “An Empirical Study on Software Testing Practices in Automotive.” en. In: *Software Process Improvement and Capability Determination*. Ed. by P. M. Clarke, R. V. O’Connor, T. Rout, A. Dorling. Cham: Springer International Publishing, 2016, pp. 301–315. ISBN: 978-3-319-38980-6. DOI: 10.1007/978-3-319-38980-6_22 (cit. on p. 31).

[LDP22] R. López-Viana, J. Díaz, J. E. Pérez. “Continuous Deployment in IoT Edge Computing : A GitOps implementation.” In: *2022 17th Iberian Conference on Information Systems and Technologies (CISTI)*. ISSN: 2166-0727. June 2022, pp. 1–6. DOI: 10.23919/CISTI54924.2022.9820108. URL: https://ieeexplore.ieee.org/abstract/document/9820108 (visited on 11/06/2024) (cit. on pp. 16, 33, 34, 39, 44).

[LK21] M. Lahami, M. Krichen. “A survey on runtime testing of dynamically adaptable and distributed systems.” en. In: *Software Quality Journal* 29.2 (June 2021), pp. 555–593. ISSN: 1573-1367. DOI: 10.1007/s11219-021-09558-x. URL: https://doi.org/10.1007/s11219-021-09558-x (visited on 01/02/2025) (cit. on p. 32).

[LL90] L. Lamport, N. Lynch. “CHAPTER 18 - Distributed Computing: Models and Methods.” In: *Formal Models and Semantics*. Ed. by J. Van leeuwen. Handbook of Theoretical Computer Science. Amsterdam: Elsevier, Jan. 1990, pp. 1157–1199. ISBN: 978-0-444-88074-1. DOI: 10.1016/B978-0-444-88074-1.50023-8. URL: https://www.sciencedirect.com/science/article/pii/B9780444880741500238 (visited on 01/05/2025) (cit. on pp. 22–24).

[LNP+18] M. Le Pallec, L. Noirie, P. Peloso, D. T. Bui, N. Le Sauze. “Digital assistance for the automated discovery and deployment of IoT services.” In: *2018 21st Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN)*. ISSN: 2472-8144. Feb. 2018, pp. 1–3. DOI: 10.1109/ICIN.2018.8401599. URL: https://ieeexplore.ieee.org/abstract/document/8401599 (visited on 11/06/2024) (cit. on p. 28).

[LS13] R. Lachmann, I. Schaefer. “Herausforderungen beim Testen von Fahrerassistenzsystemen.” de. In: Gesellschaft für Informatik e.V., 2013, pp. 2473–2487. ISBN: 978-3-88579-614-5. URL: https://dl.gi.de/handle/20.500.12116/20672 (visited on 01/29/2025) (cit. on p. 31).

[LS17] M. Lehmann, F. E. Sandnes. “A framework for evaluating continuous microservice delivery strategies.” In: *Proceedings of the Second International Conference on Internet of things, Data and Cloud Computing*. ICC ’17. New York, NY, USA: Association for Computing Machinery, Mar. 2017, pp. 1–9. ISBN: 978-1-4503-4774-7. DOI: 10.1145/3018896.3018961. URL: https://dl.acm.org/doi/10.1145/3018896.3018961 (visited on 03/24/2025) (cit. on pp. 67, 68).

[LVCD13] F. Li, M. Vögler, M. Claeßens, S. Dustdar. “Towards Automated IoT Application Deployment by a Cloud-Based Approach.” In: *2013 IEEE 6th International Conference on Service-Oriented Computing and Applications*. ISSN: 2163-2871. Dec. 2013, pp. 61–68. DOI: 10.1109/SOCA.2013.12. URL: https://ieeexplore.ieee.org/document/6717286/?arnumber=6717286 (visited on 01/06/2025) (cit. on pp. 26–28).

[LZZ16] B. Liu, H. Zhang, S. Zhu. “An Incremental V-Model Process for Automotive Development.” In: *2016 23rd Asia-Pacific Software Engineering Conference (APSEC)*. ISSN: 1530-1362. Dec. 2016, pp. 225–232. DOI: 10.1109/APSEC.2016.040. URL: https://ieeexplore.ieee.org/document/7890592/ (visited on 04/13/2025) (cit. on p. 15).

[MAA20] W. M. Eid, S. Atawneh, M. Al-Akhras. “Framework for Cybersecurity Centers to Mass Scan Networks.” en. In: *Intelligent Automation & Soft Computing* 26.4 (2020), pp. 1319–1334. ISSN: 1079-8587. DOI: 10.32604/iasc.2020.013678. URL: https://www.techscience.com/iasc/v26n6/41026 (visited on 03/10/2025) (cit. on p. 56).

[Mar21] P. Marwedel. *Embedded System Design: Embedded Systems Foundations of Cyber-Physical Systems, and the Internet of Things*. English. Accepted: 2021-02-15T10:20:43Z. Springer Nature, 2021. ISBN: 978-3-030-60910-8. DOI: 10.1007/978-3-030-60910-8. URL: https://library.oapen.org/handle/20.500.12657/46817 (visited on 01/06/2025) (cit. on pp. 21, 22).

[MB20] S. Mysari, V. Bejgam. “Continuous Integration and Continuous Deployment Pipeline Automation Using Jenkins Ansible.” In: *2020 International Conference on Emerging Trends in Information Technology and Engineering (ic-ETITE)*. Feb. 2020, pp. 1–4. DOI: 10.1109/ic-ETITE47903.2020.239. URL: https://ieeexplore.ieee.org/document/9077670/?arnumber=9077670 (visited on 11/19/2024) (cit. on pp. 15, 29, 35, 39, 43).

[MLS+23] S. R. Mallick, R. K. Lenka, S. Sudershana, A. Sahoo, S. Palei, R. K. Barik. "An Investigation Into the Efficacy of RANOREX Software Test Automation Tool." In: *2023 3rd International Conference on Innovative Sustainable Computational Technologies (CISCT)*. Sept. 2023, pp. 1–5. DOI: 10.1109/CISCT57197.2023.10351231. URL: https://ieeexplore.ieee.org/document/10351231 (visited on 03/18/2025) (cit. on pp. 67, 68).

[MMBT19] N. Medhat, S. Moussa, N. Badr, M. F. Tolba. "Testing Techniques in IoT-based Systems." In: *2019 Ninth International Conference on Intelligent Computing and Information Systems (ICICIS)*. Dec. 2019, pp. 394–401. DOI: 10.1109/ICICIS46948.2019.9014711. URL: https://ieeexplore.ieee.org/document/9014711/?arnumber=9014711 (visited on 11/06/2024) (cit. on p. 32).

[MSEM24] J. B. Minani, F. Sabir, Y. El Fellah, N. Moha. "Towards an Automated Approach for Testing IoT Devices." In: *Proceedings of the ACM/IEEE 6th International Workshop on Software Engineering Research & Practices for the Internet of Things*. SERP4IoT '24. New York, NY, USA: Association for Computing Machinery, Aug. 2024, pp. 22–29. ISBN: 979-8-4007-0578-6. DOI: 10.1145/3643794.3648350. URL: https://dl.acm.org/doi/10.1145/3643794.3648350 (visited on 11/06/2024) (cit. on p. 30).

[MTK+25] B. R. Mudhivarthi, P. Thakur, A. Kumar, S. Anturkar, G. Singh. "Firmware Over the Air for Securely Updating ECUs of the Vehicle." en. In: (Jan. 2025). Publisher: Bentham Science Publishers. DOI: 10.2174/0122103279346608250109112823. URL: https://www.benthamdirect.com/content/journals/swcc/10.2174/0122103279346608250109112823 (visited on 04/13/2025) (cit. on p. 15).

[MZA18] A. Mateen, Q. Zhu, S. Afsar. "Comparitive Analysis of Manual vs Automotive Testing for Software Quality." In: *Proceedings of the 7th International Conference on Software Engineering and New Technologies*. ICSENT 2018. New York, NY, USA: Association for Computing Machinery, Dec. 2018, pp. 1–7. ISBN: 978-1-4503-6101-9. DOI: 10.1145/3330089.3330121. URL: https://dl.acm.org/doi/10.1145/3330089.3330121 (visited on 03/18/2025) (cit. on pp. 67, 68).

[NHB05] T. Nolte, H. Hansson, L. Bello. "Automotive communications-past, current and future." In: *2005 IEEE Conference on Emerging Technologies and Factory Automation*. Vol. 1. ISSN: 1946-0759. Sept. 2005, 8 pp.–992. DOI: 10.1109/ETFA.2005.1612631. URL: https://ieeexplore.ieee.org/document/1612631/?arnumber=1612631 (visited on 12/26/2024) (cit. on p. 49).

[Nin23] A. Ning. "An Ansible-based Distributed Application Architecture Rapid Deployment Scheme." In: *2023 IEEE 2nd International Conference on Electrical Engineering, Big Data and Algorithms (EEBDA)*. Feb. 2023, pp. 972–975. DOI: 10.1109/EEBDA56825.2023.10090753. URL: https://ieeexplore.ieee.org/document/10090753 (visited on 01/15/2025) (cit. on p. 29).

[NK05] J. Novak, P. Kocourek. "Automated Testing of Electronic Control Units Compatibility in Vehicle CAN Networks." en. In: *Proceedings of the IEEE International Symposium on Industrial Electronics, 2005. ISIE 2005*. Dubrovnik, Croatia: IEEE, 2005, pp. 1423–1428. ISBN: 978-0-7803-8738-6. DOI: 10.1109/ISIE.2005.1529141. URL: http://ieeexplore.ieee.org/document/1529141/ (visited on 12/03/2024) (cit. on pp. 35, 36, 39, 43).

[NSSW05] N. Navet, Y. Song, F. Simonot-Lion, C. Wilwert. "Trends in Automotive Communication Systems." In: *Proceedings of the IEEE* 93.6 (June 2005). Conference Name: Proceedings of the IEEE, pp. 1204–1223. ISSN: 1558-2256. DOI: 10.1109/JPROC.2005.849725. URL: https://ieeexplore.ieee.org/document/1435746/?arnumber=1435746 (visited on 12/26/2024) (cit. on p. 49).

[PLF18a] P. M. Pontes, B. Lima, J. P. Faria. "Izinto: a pattern-based IoT testing framework." In: *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops*. ISSTA '18. New York, NY, USA: Association for Computing Machinery, July 2018, pp. 125–131. ISBN: 978-1-4503-5939-9. DOI: 10.1145/3236454.3236511. URL: https://dl.acm.org/doi/10.1145/3236454.3236511 (visited on 11/06/2024) (cit. on p. 30).

[PLF18b] P. M. Pontes, B. Lima, J. P. Faria. "Test patterns for IoT." In: *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*. A-TEST 2018. New York, NY, USA: Association for Computing Machinery, Nov. 2018, pp. 63–66. ISBN: 978-1-4503-6053-1. DOI: 10.1145/3278186.3278196. URL: https://dl.acm.org/doi/10.1145/3278186.3278196 (visited on 11/06/2024) (cit. on p. 32).

[Pra19] R. Prashanth. "Improving the Flashing Speed of Electronic Control Unit in Automotive Industries." In: *2019 Innovations in Power and Advanced Computing Technologies (i-PACT)*. Vol. 1. Mar. 2019, pp. 1–6. DOI: 10.1109/i-PACT44901.2019.8960182. URL: https://ieeexplore.ieee.org/document/8960182/?arnumber=8960182 (visited on 12/03/2024) (cit. on p. 49).

[San16] B. Sand. "IoT Testing - The Big Challenge Why, What and How." en. In: *Internet of Things. IoT Infrastructures*. Ed. by B. Mandler, J. Marquez-Barja, M. E. Mitre Campista, D. Cagáňová, H. Chaouchi, S. Zeadally, M. Badra, S. Giordano, M. Fazio, A. Somov, R.-L. Vieriu. Cham: Springer International

Publishing, 2016, pp. 70–76. ISBN: 978-3-319-47075-7. DOI: 10.1007/978-3-319-47075-7_9 (cit. on p. 32).

[SEL19] Y. Sabri, N. El Kamoun, F. Lakrami. "A Survey: Centralized, Decentralized, and Distributed Control Scheme in Smart Grid Systems." In: *2019 7th Mediterranean Congress of Telecommunications (CMT)*. Oct. 2019, pp. 1–11. DOI: 10.1109/CMT.2019.8931370. URL: https://ieeexplore.ieee.org/abstract/document/8931370?casa_token=kiXFPodJZmQAAAAA:YS9BvY9jY-mECIKEX_Vret52zJkWj9IZeqjgjNTKtDqW1H1kS37HxHhRKdXIRAVuQnpK4H7r2 (visited on 02/18/2025) (cit. on p. 23).

[SFF+24] J. Sánchez-Garrido, E. Florentín, M. Fuentes, T. García, G. Ros, R. Torrego. "Integrating the synchronization stacks of a heterogeneous time-sensitive networking system with wired and wireless domains." In: *2024 24th International Conference on Transparent Optical Networks (ICTON)*. ISSN: 2161-2064. July 2024, pp. 1–4. DOI: 10.1109/ICTON62926.2024.10647531. URL: https://ieeexplore.ieee.org/abstract/document/10647531?casa_token=_o5grdoTA2EAAAAA:S7rO4eLyh6ruIVAFrKJlL3U0HA_rkHvOahIj6Jof9Z-2BdZIkCqvyurieWIASiQxXjfffEANDy_r (visited on 02/03/2025) (cit. on p. 32).

[SGP+19] D. Serra, G. Grano, F. Palomba, F. Ferrucci, H. C. Gall, A. Bacchelli. "On the Effectiveness of Manual and Automatic Unit Test Generation: Ten Years Later." In: *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. ISSN: 2574-3864. May 2019, pp. 121–125. DOI: 10.1109/MSR.2019.00028. URL: https://ieeexplore.ieee.org/document/8816768 (visited on 03/18/2025) (cit. on pp. 67, 68).

[SM20] B. Souza, P. Machado. "A Large Scale Study On the Effectiveness of Manual and Automatic Unit Test Generation." In: *Proceedings of the XXXIV Brazilian Symposium on Software Engineering*. SBES '20. New York, NY, USA: Association for Computing Machinery, Dec. 2020, pp. 253–262. ISBN: 978-1-4503-8753-8. DOI: 10.1145/3422392.3422407. URL: https://dl.acm.org/doi/10.1145/3422392.3422407 (visited on 03/18/2025) (cit. on p. 67).

[SSR22] N. Singh, A. Singh, V. Rawat. "Deploying Jenkins, Ansible and Kubernetes to Automate Continuous Integration and Continuous Deployment Pipeline." In: *2022 IEEE International Conference on Service Operations and Logistics, and Informatics (SOLI)*. ISSN: 2768-1890. Dec. 2022, pp. 1–5. DOI: 10.1109/SOLI57430.2022.10294378. URL: https://ieeexplore.ieee.org/document/10294378/?arnumber=10294378 (visited on 11/19/2024) (cit. on pp. 29, 35, 39, 44).

[SV18] E. C. dos Santos, P. Vilain. "Automated Acceptance Tests as Software Requirements: An Experiment to Compare the Applicability of Fit Tables and Gherkin Language." In: *Agile Processes in Software Engineering and Extreme Programming*. Ed. by J. Garbajosa, X. Wang, A. Aguiar. Cham: Springer International Publishing, 2018, pp. 104–119. ISBN: 978-3-319-91602-6 (cit. on p. 51).

[TDS+13] S. Thummalapenta, P. Devaki, S. Sinha, S. Chandra, S. Gnanasundaram, D. D. Nagaraj, S. Kumar, S. Kumar. "Efficient and change-resilient test automation: An industrial case study." In: *2013 35th International Conference on Software Engineering (ICSE)*. ISSN: 1558-1225. May 2013, pp. 1002–1011. DOI: 10.1109/ICSE.2013.6606650. URL: https://ieeexplore.ieee.org/document/6606650 (visited on 03/18/2025) (cit. on pp. 30, 67, 68).

[TGBG20] C. Tsigkanos, M. Garriga, L. Baresi, C. Ghezzi. "Cloud Deployment Tradeoffs for the Analysis of Spatially Distributed Internet of Things Systems." en. In: *ACM Transactions on Internet Technology* 20.2 (May 2020), pp. 1–23. ISSN: 1533-5399, 1557-6051. DOI: 10.1145/3381452. URL: https://dl.acm.org/doi/10.1145/3381452 (visited on 01/15/2025) (cit. on p. 27).

[TJLG17] S. Tata, R. Jain, H. Ludwig, S. Gopisetty. "Living in the Cloud or on the Edge: Opportunities and Challenges of IOT Application Architecture." In: *2017 IEEE International Conference on Services Computing (SCC)*. ISSN: 2474-2473. June 2017, pp. 220–224. DOI: 10.1109/SCC.2017.35. URL: https://ieeexplore.ieee.org/document/8034988 (visited on 01/15/2025) (cit. on p. 28).

[TS07] A. S. Tanenbaum, M. v. Steen. *Distributed systems: principles and paradigms*. en. 2nd ed. OCLC: ocm70707891. Upper Saddle RIiver, NJ: Pearson Prentice Hall, 2007. ISBN: 978-0-13-239227-3 (cit. on pp. 21–25, 47).

[UK99] A. Ulrich, H. König. "Architectures for Testing Distributed Systems." en. In: *Testing of Communicating Systems: Methods and Applications*. Ed. by G. Csopaki, S. Dibuz, K. Tarnay. Boston, MA: Springer US, 1999, pp. 93–108. ISBN: 978-0-387-35567-2. DOI: 10.1007/978-0-387-35567-2_7. URL: https://doi.org/10.1007/978-0-387-35567-2_7 (visited on 01/07/2025) (cit. on p. 30).

[VBO+16] A. M. R. Vincenzi, T. Bachiega, D. G. de Oliveira, S. R. S. de Souza, J. C. Maldonado. "The complementary aspect of automatically and manually generated test case sets." In: *Proceedings of the 7th International Workshop on Automating Test Case Design, Selection, and Evaluation*. A-TEST 2016. New York, NY, USA: Association for Computing Machinery, Nov. 2016, pp. 23–30. ISBN: 978-1-4503-4401-2. DOI: 10.1145/2994291.2994295.

URL: https://dl.acm.org/doi/10.1145/2994291.2994295 (visited on 03/18/2025) (cit. on pp. 67, 68).

[VPBB22] M. Viggiato, D. Paas, C. Buzon, C.-P. Bezemer. “Using natural language processing techniques to improve manual test case descriptions.” In: *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*. ICSE-SEIP ’22. New York, NY, USA: Association for Computing Machinery, Oct. 2022, pp. 311–320. ISBN: 978-1-4503-9226-6. DOI: 10.1145/3510457.3513045. URL: https://dl.acm.org/doi/10.1145/3510457.3513045 (visited on 03/18/2025) (cit. on p. 31).

[WS14] S. Weißleder, H. Schlingloff. “An Evaluation of Model-Based Testing in Embedded Applications.” In: *Verification and Validation 2014 IEEE Seventh International Conference on Software Testing*. ISSN: 2159-4848. Mar. 2014, pp. 223–232. DOI: 10.1109/ICST.2014.35. URL: https://ieeexplore.ieee.org/document/6823884 (visited on 03/18/2025) (cit. on pp. 67, 68).

[WSG98] T. Walter, I. Schieferdecker, J. Grabowski. “Test Architectures for Distributed Systems — State of the Art and Beyond.” en. In: *Testing of Communicating Systems: Proceedings of the IFIP TC6 11th International Workshop on Testing of Communicating Systems (IWTCS’98) August 31-September 2, 1998, Tomsk, Russia*. Ed. by A. Petrenko, N. Yevtushenko. Boston, MA: Springer US, 1998, pp. 149–174. ISBN: 978-0-387-35381-4. DOI: 10.1007/978-0-387-35381-4_10. URL: https://doi.org/10.1007/978-0-387-35381-4_10 (visited on 01/07/2025) (cit. on pp. 30, 32).

[WTS03] J. P. Wack, M. C. Tracy, M. P. Souppaya. *Guideline on network security testing*. en. Tech. rep. NIST SP 800-42. Edition: 0. Gaithersburg, MD: National Institute of Standards and Technology, 2003, NIST SP 800–42. DOI: 10.6028/NIST.SP.800-42. URL: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-42.pdf (visited on 03/10/2025) (cit. on p. 56).

[YBB06] J. Yang, J. Bauman, A. Beydoun. *An Effective Model-Based Development Process Using Simulink/Stateflow for Automotive Body Control Electronics*. English. SAE Technical Paper 2006-01-3501. ISSN: 0148-7191, 2688-3627. Warrendale, PA: SAE International, Oct. 2006. DOI: 10.4271/2006-01-3501. URL: https://www.sae.org/publications/technical-papers/content/2006-01-3501/ (visited on 03/12/2025) (cit. on p. 43).

[YTY18] W. Yiran, Z. Tongyang, G. Yidong. “Design and implementation of continuous integration scheme based on Jenkins and Ansible.” In: *2018 International Conference on Artificial Intelligence and Big Data (ICAIBD)*. May 2018, pp. 245–249. DOI: 10.1109/ICAIBD.2018.8396203. URL: https:

//ieeexplore.ieee.org/document/8396203/?arnumber=8396203 (visited on 11/19/2024) (cit. on p. 29).

[ZSA+14] I. Zhang, A. Szekeres, D.V. Aken, I. Ackerman, S.D. Gribble, A. Krishnamurthy, H.M. Levy. "Customizable and Extensible Deployment for {Mobile/Cloud} Applications." en. In: 2014, pp. 97–112. ISBN: 978-1-931971-16-4. URL: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/zhang (visited on 01/17/2025) (cit. on pp. 24, 25, 27).

[ZZT10] W. Zhuang, Y. Zhang, H. Tan. "A Decentralized Control Algorithm for Maintaining Communication Links in Mobile Sensor Networks." In: *2010 International Conference on Intelligent Computation Technology and Automation*. Vol. 1. May 2010, pp. 61–64. DOI: 10.1109/ICICTA.2010.224. URL: https://ieeexplore.ieee.org/document/5522829 (visited on 01/15/2025) (cit. on pp. 21–25).

All links were last followed on April 22, 2025.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

place, date, signature