universität wien

DISSERTATION | DOCTORAL THESIS

Titel | Title

Systematic MLOps and RLOps Engineering: Architectural Knowledge, Comprehension Models, Conformance Assessment and Pipeline Generation

verfasst von | submitted by

Stephen John Warnett BSc (Hons) MSc

angestrebter akademischer Grad | in partial fulfilment of the requirements for the degree of

Doktor der technischen Wissenschaften (Dr.techn.)

Wien | Vienna, 2026

Studienkennzahl lt. Studienblatt | Degree programme code as it appears on the student record sheet:	UA 786 880
Dissertationsgebiet lt. Studienblatt | Field of study as it appears on the student record sheet:	Informatik
Betreut von | Supervisor:	Univ.-Prof. Dr. Uwe Zdun

Acknowledgements

I want to express my sincere gratitude to my doctoral advisor, Uwe Zdun, for the opportunity to conduct my doctoral research within the Research Group Software Architecture. I am grateful for his invaluable guidance, advice and support throughout my doctoral journey.

I would also like to thank my other colleagues in the Research Group Software Architecture. Thanks go particularly to Evangelos Ntentos for our work together, Sylvia Ennsberger for organisational support, and Gerhard Pernecker for technical support.

Further thanks are extended to the UniVie Doctoral School Computer Science DoCS, and the funding agencies that helped facilitate this research – the Austrian Research Promotion Agency (FFG) and the Austrian Science Fund (FWF). I would also like to thank and acknowledge our project partners. In particular, I thank Sebastian Geiger and his colleagues at Siemens Aktiengesellschaft Österreich for our collaboration.

Controlled experiments conducted over the course of the doctoral research would not have been possible without the participation of students and tutors. I thank them for their engagement.

The research papers upon which this doctoral thesis is based were greatly enhanced via the peer review process. I extend my thanks to the numerous anonymous peers who reviewed and provided feedback on our various journal, conference and workshop submissions. Similarly, I wish to thank the examiners for agreeing to review this doctoral thesis.

Finally, I would like to express my deepest gratitude and appreciation to my family, who accompanied me every step of the way. This doctoral thesis would not have been possible without the support, advice and patience of my wife, Anna, and our children, Dominik, Matilda and Julia. I dedicate this work to you.

Abstract

Machine learning operations (MLOps) and reinforcement learning operations (RLOps) have established themselves as central disciplines for the operation and maintenance of machine learning (ML) and reinforcement learning (RL) models in production systems. However, the field faces several interrelated challenges that affect the maturity and accessibility of MLOps and RLOps practices.

Firstly, practitioners lack systematic guidance on the necessary architectural design decisions (ADDs) for designing effective MLOps and RLOps systems, with existing knowledge remaining largely implicit and undocumented. Secondly, this lack of architectural knowledge applies in particular to RL in specialised contexts such as Industry 4.0 cyber-physical production systems (CPPSs) and to the emerging practice of RLOps. Thirdly, the complexity of MLOps and RL system architectures creates barriers to understanding, particularly for developers with limited ML and RL experience, which hinders knowledge transfer and collaborative design. Fourthly, ensuring compliance with ADDs, patterns, practices and decision options in MLOps and RLOps systems is a time-consuming and error-prone manual task. Objective measures – such as metrics – for evaluating architectural decision options are also lacking, and there is a need for standardised, automated procedures to detect compliance with best practices, particularly regarding training methods for RL. Fifthly, the manual effort involved in designing, configuring and validating MLOps and RLOps pipelines is labour-intensive and error-prone, creating barriers for practitioners without specialist DevOps expertise and limiting the democratisation of ML and RL adoption.

This doctoral thesis addresses these challenges through comprehensive studies ranging from the extraction of architectural knowledge to quality assessment and automation. We systematically identify and formalise the most important ADDs, their decision options, relationships and decision drivers for ML and RL systems, using qualitative Straussian Grounded Theory that builds on the knowledge of practitioners. In doing so, we document how MLOps and RLOps exhibit distinct architectural patterns and how RL-specific requirements necessitate adaptations to general MLOps decisions when applied to CPPSs. Through empirical studies, we investigate how architectural models and documentation influence developers' understanding of MLOps and RL systems, and we develop a model-driven, metrics-based framework for systematically assessing whether MLOps systems adequately support critical quality aspects such as automation. Furthermore, we explore the use of large language models (LLMs) as an evaluation approach to identify compliance with best practices for RL training pipelines and compare their effectiveness with traditional heuristic-based code detectors. Finally, we demonstrate how LLMs can reliably automate the generation of correct MLOps and RLOps pipeline configurations using low-code, template-based, modular approaches, and empirically evaluate several

LLMs in general RL contexts as well as in specialised Industry 4.0 environments.

Our results show that: (1) ML workflows and deployment comprise 16 key ADDs with interlinked relationships and domain-specific decision drivers; (2) RL architectures introduce six key ADDs, with specific patterns relating to training strategies, agent configuration and deployment. An industry case study confirms which parts of the ML/RL ADD catalogue were adopted unchanged, which required RL-specific adaptations, which were not applied and which were not applicable; (3) semi-formal architectural models significantly improve practitioners' understanding of MLOps systems and their task performance, with understanding of the RL source code improving in specific contexts; (4) systematic quality assessment frameworks that employ both metrics and architectural rules effectively identify best practice compliance in MLOps and RLOps systems; (5) LLMs can generate correct MLOps pipeline configurations for RL with initial error rates as low as 0.09 (GPT-4o), and error-free generation is achievable following targeted improvements; and LLM-based pipeline generation approaches can be reliably transferred to specialised Industry 4.0 RLOps contexts with perfect results, reducing development time and eliminating the need for specialised DevOps expertise.

Our work makes MLOps and RLOps more accessible, understandable and practical across various organisational contexts. It provides a comprehensive formal ADD model for ML and RL systems, delivers empirical evidence for the benefits of semi-formal architectural documentation, develops a model-driven framework for quality assessment in MLOps, employs LLMs for compliance-based architectural evaluation in RL systems, and offers LLM-based solutions for automated, reliable pipeline generation. Taken together, these contributions advance the maturity of ML/RL system engineering and demonstrate how emerging artificial intelligence technologies can automate pipeline generation.

Kurzfassung

Machine learning operations (MLOps) und reinforcement learning operations (RLOps) haben sich als zentrale Disziplinen für den Betrieb und die Wartung von Machine-Learning- (ML) und Reinforcement-Learning- (RL) Modellen in produktiven Systemen etabliert. Das Feld ist jedoch mit mehreren miteinander verknüpften Herausforderungen konfrontiert, die die Reife und Zugänglichkeit der MLOps- und RLOps-Praxis beeinträchtigen.

Erstens fehlt es Praktikerinnen und Praktikern an einer systematischen Orientierung zu den erforderlichen architektonischen Designentscheidungen (ADDs) für die Gestaltung wirkungsvoller MLOps- und RLOps-Systeme, wobei vorhandenes Wissen weitgehend implizit und undokumentiert bleibt. Zweitens gilt dieser Mangel an architektonischem Wissen insbesondere für RL in spezialisierten Kontexten wie Industrie-4.0-cyber-physischen Produktionssystemen und für die sich herausbildende Praxis von RLOps. Drittens führt die Komplexität von MLOps- und RL-Systemarchitekturen zu Verständnishürden, insbesondere für Entwicklerinnen und Entwickler mit begrenzter ML- und RL-Erfahrung, was den Wissenstransfer und die kollaborative Gestaltung erschwert. Viertens ist die Sicherstellung der Konformität mit ADDs, Mustern, Praktiken und Entscheidungsoptionen in MLOps- und RLOps-Systemen eine aufwändige, fehleranfällige manuelle Aufgabe. Objektive Messgrößen – etwa Metriken – zur Bewertung architektonischer Entscheidungsoptionen fehlen ebenfalls, und es besteht Bedarf an standardisierten, automatisierten Verfahren zur Erkennung der Konformität mit Best Practices, insbesondere hinsichtlich der Trainingsmethoden für RL. Fünftens ist der manuelle Aufwand für die Gestaltung, Konfiguration und Validierung von MLOps- und RLOps-Pipelines arbeitsintensiv und fehleranfällig, schafft Hürden für Praktikerinnen und Praktiker ohne spezialisierte DevOps-Expertise und begrenzt die Demokratisierung des ML- und RL-Einsatzes.

Diese Dissertation adressiert diese Herausforderungen durch umfassende Studien, die von der Erschließung architektonischen Wissens über die Qualitätsbewertung bis hin zur Automatisierung reichen. Wir identifizieren und formalisieren systematisch die wichtigsten ADDs, ihre Entscheidungsoptionen, Beziehungen und Entscheidungsfaktoren für ML- und RL-Systeme mithilfe einer qualitativen, auf Strauss basierenden Grounded Theory, die auf dem Wissen von Praktikern aufbaut. Dabei dokumentieren wir, wie MLOps und RLOps unterschiedliche architektonische Muster aufweisen und wie RL-spezifische Anforderungen Anpassungen allgemeiner MLOps-Entscheidungen erfordern, wenn sie für cyber-physische Produktionssysteme angewendet werden. Wir untersuchen in empirischen Studien, wie architektonische Modelle und Dokumentation das Verständnis von Entwicklerinnen und Entwicklern für MLOps- und RL-Systeme beeinflussen, und entwickeln ein modellgetriebenes, metrikenbasiertes Rahmenwerk zur systematischen Bewertung, ob MLOps-Systeme kritische Qualitätsaspekte wie die Automatisierung hinreichend unterstützen. Darüber hinaus erforschen wir den Einsatz von Large Language

Models (LLMs) als Bewertungsansatz zur Identifikation der Konformität mit Best Practices für RL-Trainingspipelines und vergleichen deren Wirksamkeit mit traditionellen heuristikbasierten Code-Detektoren. Abschließend zeigen wir, wie LLMs die Generierung korrekter MLOps- und RLOps-Pipelinekonfigurationen mit Low-Code-, templatebasierten, modularen Ansätzen verlässlich automatisieren können, und evaluieren mehrere LLMs empirisch in allgemeinen RL-Kontexten sowie in spezialisierten Industrie-4.0-Umgebungen.

Unsere Ergebnisse zeigen, dass: (1) ML-Workflows und Deployment 16 zentrale ADDs mit miteinander verknüpften Beziehungen und domänenspezifischen Entscheidungsfaktoren umfassen; (2) RL-Architekturen führen sechs zentrale ADDs ein, mit speziellen Mustern in Bezug auf Trainingsstrategien, Agentenkonfiguration und -einsatz. Eine Fallstudie aus der Industrie bestätigt, welche Teile des ML/RL-ADD-Katalogs unverändert übernommen wurden, welche RL-spezifische Anpassungen erforderten, welche nicht angewendet wurden und welche nicht anwendbar waren; (3) semi-formale Architekturmodelle das MLOps-Systemverständnis und die Aufgabenerfüllung von Praktikerinnen und Praktikern signifikant verbessern, wobei sich das Verständnis des RL-Quellcodes in bestimmten Kontexten verbessert; (4) systematische Qualitätsbewertungsrahmen, die sowohl Metriken als auch architektonische Regeln einsetzen, Best-Practice-Konformität in MLOps- und RLOps-Systemen wirksam identifizieren; (5) LLMs korrekte MLOps-Pipelinekonfigurationen für RL mit anfänglichen Fehlerraten von nur 0,09 (GPT-4o) erzeugen können und nach gezielten Verbesserungen eine fehlerfreie Generierung erreichbar ist, und LLM-basierte Pipelinegenerierungsansätze sich mit perfekten Ergebnissen zuverlässig auf spezialisierte Industrie-4.0-RLOps-Kontexte übertragen lassen, Entwicklungszeit reduzieren und die Notwendigkeit spezialisierter DevOps-Kenntnisse eliminieren.

Unsere Arbeit macht MLOps und RLOps in verschiedenen organisatorischen Kontexten zugänglicher, verständlicher und praktikabler. Sie bietet ein umfassendes formales ADD-Modell für ML- und RL-Systeme, liefert empirische Belege für die Vorteile einer semiformalen Architekturdokumentation, entwickelt ein modellgetriebenes Framework für die Qualitätsbewertung in MLOps, nutzt LLMs zur regelkonformen Architekturbewertung in RL-Systemen und bietet LLM-basierte Lösungen für die automatisierte, zuverlässige Pipeline-Generierung. Zusammengenommen erhöhen diese Beiträge die Reife des ML/RL-System-Engineerings und zeigen, wie Pipelines mithilfe neuer Technologien der künstlichen Intelligenz automatisiert generiert werden können.

Contents

List of Tables

List of Figures

List of Listings

Part I

Introduction

1 Thesis Overview and Research Methodology

1.1 Introduction and Key Concepts

The objective of this thesis is the study of machine learning (ML) and reinforcement learning (RL)-based systems [1], [2], [3] and the development of tools and techniques to better support practitioners systematically in the domains of architectural design, comprehension models, quality assessment and automated pipeline generation for ML and RL operations [4], [5], [6], [7], [8], [9]. It details the development of techniques to systematically reduce manual labour in the productionisation of ML and RL models and applies these prototypical techniques to a real-world Industry 4.0 [10], [11], [12] cyber-physical production system [13], [14], [15], [16].

These core concepts and terminology – architectural design decisions, ML operations, RL operations, large language models, Industry 4.0 and cyber-physical production systems – are used throughout this doctoral thesis and are described below. Incidental concepts will be explained as necessary throughout the thesis.

Architectural design decisions (ADDs). ADDs are the decisions upon which software architecture is based. ADDs are essential in software architecture because they capture the architectural choices made during the early design process. ADDs have a long-term impact on the system in question and are costly to change. Architects need to prioritise, as architectural decisions involve trade-offs amongst requirements [17], [18].

In architectural design, some critical ADDs form the basis upon which a software system is constructed. These ADDs establish the architecture and layout of the system and its components and their integration, alongside highly influential quality attributes such as speed, extensibility, modularity, security and modifiability. Of significance in ADDs are *relations*, which refer to connections and interactions between system components [17]. These relations reflect functionality, communication and data flow.

Software architects must consider several factors when determining the system's architectural design. These factors include the functionality required from the system, the available environment and its capacity, necessary tools and technologies, as well as the abilities and preferences of the development team. They also need to make judgments on the relative costs and benefits of various decision options and their impacts on quality characteristics, such as performance, reliability and security [19]. These factors can be considered *decision drivers* (or *forces*) in the context of ADDs, encompassing the dynamic aspects and constraints that shape the decisions. These aspects can be external

or internal, spanning quality, functional and non-functional requirements, technological limitations, organisational policies and market conditions. Software architects balance the trade-offs between forces to achieve desired outcomes, addressing stakeholder needs effectively through informed decisions.

When applying ADD decision options, software architects often utilise tools and techniques that include architectural patterns, architectures and decision frameworks. They also make joint decisions with other stakeholders, including developers, project managers and end users. *Patterns* [20] refer to established guidelines, methodologies, techniques and approaches practitioners apply to make informed decisions and design effective software architectures. They are typically derived from industry best practices, architectural principles and lessons learned from previous projects and described in practitioner literature. They represent decision options that can be selected depending on preferred forces.

ML operations (MLOps). MLOps represents a fundamental paradigm in modern ML engineering and has evolved to address the complexities of deploying and maintaining ML systems in production environments. MLOps constitutes a holistic approach to the ML workflow, applying development operations (DevOps) [21], [22] and continuous integration/continuous delivery (CI/CD) [22] principles and practices, such as pipelines, to ML. It involves implementing automated, repeatable workflows for training, provisioning and managing supervised and unsupervised ML models in production environments [4], [23], [24].

MLOps is rapidly becoming an essential approach for engineering ML systems, as it defines processes and practices for models' availability and stability throughout their lifetime. Nevertheless, MLOps is still in a somewhat early stage, and there is only occasional agreement or consensus regarding what MLOps, as a relatively new set of practices, actually means [25], [26], [27], [28]. To aid understanding of the main concepts, we will describe how MLOps is defined in this work, its relation to DevOps and the deployment and management of ML systems as a whole. Please note that this section provides a high-level, general description of the concepts and terminology. We do not claim that any particular set of practices is complete or prescribe rules for how MLOps systems should be structured, nor do we prescribe links between concepts and specific practices.

Although MLOps refers specifically to ML tasks, it shares many similarities with DevOps in enhancing and automating the processes of software and ML deployment. Both adopt cross-functional collaboration and the integration and automation of processes, utilising CI/CD pipelines across multiple iterations, builds and deployments. It also conveys the basic idea of monitoring performance on an ongoing basis and providing feedback to enhance performance. However, MLOps also addresses concerns related to data, the model life cycle, model monitoring and experimentation, which DevOps typically does not. Therefore, it enables blending conventional DevOps approaches with ML-compatible workflows and practices [28].

The exemplary MLOps workflow shown in Figure 1.1 includes several key steps. The five stages typically involved are pipeline triggering, data ingestion and processing, model building, model deployment and model monitoring.

Triggers are used to automatically initiate specific actions, pipelines or workflows based

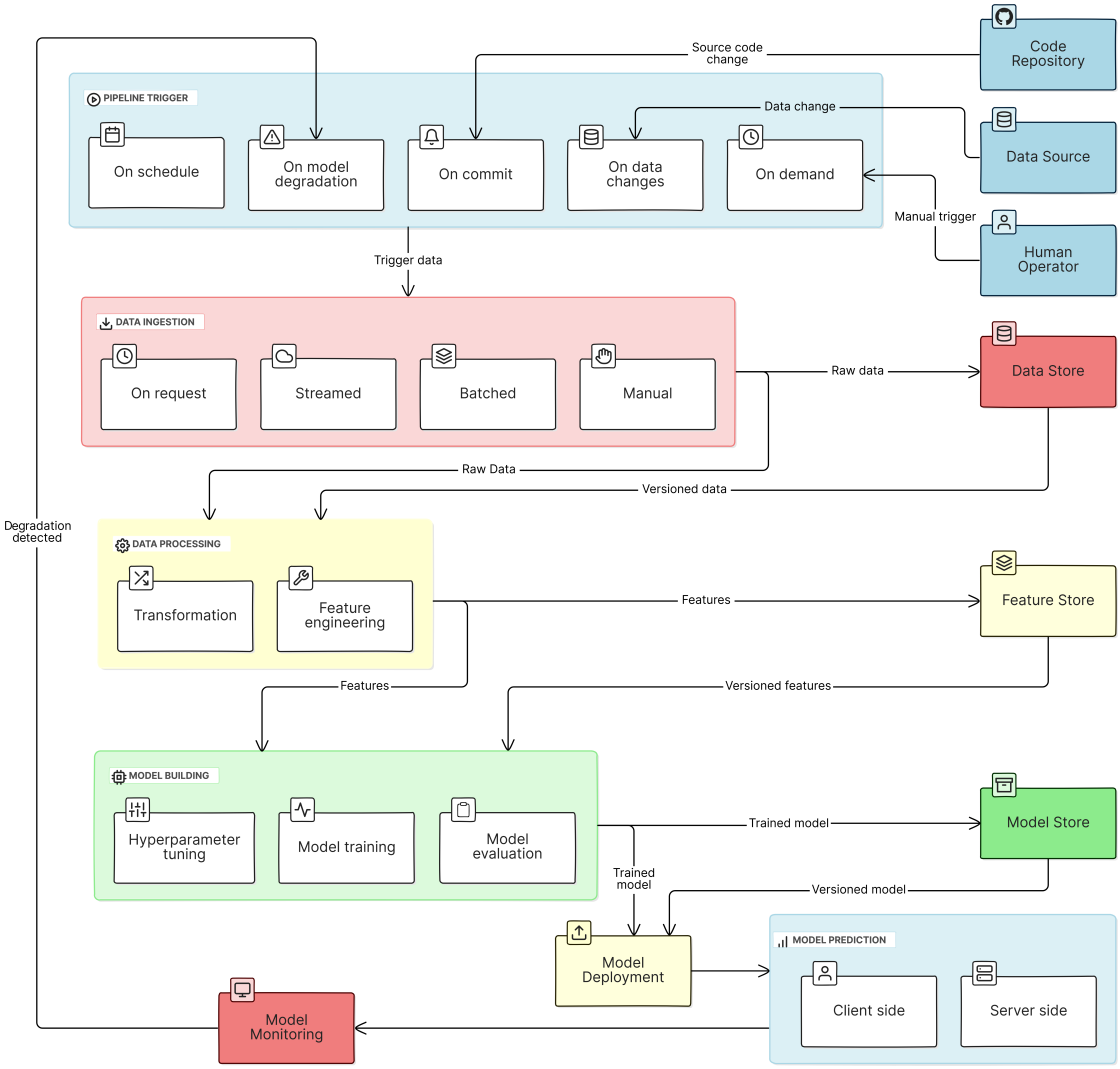

Figure 1.1: A high-level abstraction of common ML workflow tasks in MLOps.

on predefined events or conditions [29]. As in DevOps, CI/CD pipelines often utilise triggers to initiate processes – in the case of ML, to automate the ML workflow. For example, a trigger can be set up to automatically initiate data ingestion when new data becomes available in a data repository or when model performance deteriorates beyond a specified threshold.

Data ingestion is the stage where data is introduced into the system. This data may be ingested, for example, on request, automatically streamed, provided in batches or manually [30]. The raw data may be stored in a data store and then passed onto a further pipeline or pipeline step for processing.

In the data processing stage [31], the raw data is cleaned, preprocessed and transformed into a suitable format for model training. Various solutions exist for data processing,

including data pipelines, ETL pipelines and data processing components. Features for use in model training may also be engineered or extracted at this point. This stage involves preparing data suitable for model training, as poorly prepared data can degrade model performance. Feature stores store processed data and its features for training and evaluating models. Neither of these data-related steps is necessary for ML that is not based on datasets, e.g. RL.

Model building [4] involves using ML algorithms on the prepared data to identify patterns and relations. Selecting appropriate algorithms and hyperparameters can significantly impact the final model's performance, and the most suitable configuration must be determined based on the specific problem. Models are also validated at this stage. Validation is the process of evaluating a trained model's performance on previously unseen data. It involves evaluating model metrics and performance indicators, such as accuracy, precision, recall and F_1 score, to assess the trained model's suitability. Model evaluation is crucial for ensuring that trained models perform as expected and for identifying areas for improvement.

Model deployment [32] enables trained models to be used on target systems. This process involves the packaging of the model and its dependencies into a deployable artefact, such as a Docker [33] container or a serverless function. Through this process, models can be easily and consistently deployed across development, staging and production environments. Models can be delivered, for instance, via build and deployment scripts, a custom CI/CD pipeline or an ML orchestrator. For archiving and versioning trained ML models and their related artefacts, model repositories can be used. These artefacts include model files, along with related model metrics and metadata. Such model versioning enables, for example, rolling back to previous model versions when necessary.

After a model has been deployed to a production environment, continuous performance assessment through model monitoring begins, involving the collection and analysis of a model's inputs, outputs and performance metrics, to identify any performance degradation. Model monitoring helps to ensure the model continues to perform as expected over time. When tracking model performance degradation, factors such as prediction latencies, errors and model or environmental drift should also be considered. As previously mentioned, if model performance degrades, the ML workflow may be automatically triggered [29].

RL operations (RLOps) RLOps has emerged as a paradigm that is related to, but distinct from, MLOps. RLOps concerns RL and remains a critical gap in the current academic literature and industrial practice [9]. The extent to which MLOps practices apply to RL, particularly given major differences between ML and RL in model deployment and training, is addressed in this thesis.

Large language models (LLMs) LLMs are artificial intelligence (AI) models that are pre-trained on large-scale corpora using methods that do not require explicit labelling of the training data. Self-attention in transformer structures enables models to handle long sequences of data by processing them in bulk, allowing for a large number of parameters and access to large datasets [34].

Industry 4.0 and Cyber-physical production systems (CPPSs) Industry 4.0 [10], [11], [12] and CPPSs [13], [14], [15], [16] are a combination of digital technologies symbolising a significant development in modern manufacturing. Smart factories in Industry 4.0 integrate the Internet of Things, AI, and cloud computing, forming efficient networks that interact automatically and share data in real time.

1.2 Thesis Structure

The rest of **Part I** discusses the thesis's research process, including the motivation and research problems, research questions, research contributions, publications, research methods, a research overview, and a description of the overarching approach.

Part II details studies on architectural knowledge for ML and MLOps, including the ML workflow in Chapter 2 and ML deployment in Chapter 3.

Part III continues work on architectural knowledge, but this time for RL and RLOps. Specifically, Chapter 4 considers decision-making on training strategies in RL architectures, and Chapter 5 aims to bridge the gap between MLOps and RLOps with an Industry 4.0 CPPS case study.

Part IV is about understanding MLOps and RL-enabled systems. In Chapter 6, we consider the understandability of MLOps system architectures, whereas Chapter 7 investigates the understandability of ML practices in deep learning (DL) and RL-based systems.

In **Part V**, we turn our attention to assessing conformance to quality characteristics, best practices and patterns in MLOps and RL-enabled systems. Chapter 8 details a model-driven, metrics-based approach to assessing support for quality aspects in MLOps system architectures, and Chapter 9 studies the rule-based assessment of RL practices using LLMs.

Part VI utilises LLMs further and addresses pipeline generation for MLOps and RLOps using LLMs. Chapter 10 documents our initial approach to MLOps pipeline generation for RL by means of a novel low-code approach using LLMs. Chapter 11 then applies the same methodology to RLOps pipeline development with LLMs in the context of an Industry 4.0 CPPS case study with our project partner, Siemens Aktiengesellschaft Österreich.

Finally, **Part VII** concludes the thesis. Chapter 12 revisits the research questions, discusses potential enhancements to our overarching approach, considers open research challenges and then provides a summary.

1.3 Motivation and Research Problems

Transforming ML models into production-ready systems presents significant challenges. These challenges often arise from confusion and uncertainty stemming from differences between software engineering and architecture, ML methodologies and varying levels of expertise amongst practitioners [35]. Typically, ML developers lack backgrounds in software engineering or software architecture, leading them to overlook standard engineering practices. Conversely, those who specialise in software engineering and architecture seldom engage in ML engineering, which hampers their ability to devise suitable solutions

for ML systems [36], [37]. The issues resulting from this disconnect are exacerbated by technical hurdles specific to ML projects, which are typically absent in non-ML projects.

A crucial phase in the ML pipeline is model deployment. Although ML encompasses software engineering tasks such as development, maintenance, and deployment [38], [39], [40], ML practitioners have access to a variety of unique ADD options tailored to their needs. Given that the application of engineering and architectural principles in ML is relatively nascent compared to established practices in software engineering and architecture, practitioners may find it difficult to discern which options are available when making decisions, particularly during deployment. There exists a significant gap in formalised understanding and architectural frameworks for deploying ML systems.

Both ML and RL have proven to be influential methods for training intelligent agents capable of making sequential decisions. As RL architectures grow more complex, the need for informed decision-making about training strategies and their implications for software architecture becomes increasingly important, especially given the current scarcity of scientific literature on this topic. It is thus essential to improve practitioners' understanding of RL architecture strategies.

Research Problem 1 ($\mathbf{RP_1}$)

$\mathbf{RP_1}$ *Productionising ML models is technically challenging. This challenge is compounded by knowledge gaps at the individual practitioner level and the lack of formalised, reusable architectural knowledge sources. For RL, practitioners have many training, deployment and coordination strategies to choose from, and there is a need for improved understanding of architectural strategies.*

In the context of Industry 4.0, CPPSs are increasingly utilising RL as an effective means of training intelligent agents via digital twins. Whilst MLOps has been established as a comprehensive approach to automating workflows in both supervised and unsupervised ML, it is still unclear how these practices can be applied to RL, particularly given the differences in model deployment and training between ML and RL. Research on RLOps, which represents the application of MLOps principles to RL, is limited. MLOps has become a standard approach for automating workflows in supervised and unsupervised ML, encompassing data processing, model training, deployment and maintenance [4]. However, the exploration of RLOps – a growing area of interest – remains largely uncharted [7].

Research Problem 2 ($\mathbf{RP_2}$)

$\mathbf{RP_2}$ *The extent to which MLOps and architectural practices apply to RL is poorly understood.*

MLOps system descriptions in grey literature customarily comprise informal textual descriptions and informal graphical system diagrams. System architecture descriptions are often of an informal nature [41], [42], [43], such as ad-hoc box-and-line or box-and-arrow diagrams. Textual descriptions vary widely in detail and do not always match the accompanying informal MLOps system diagrams. Informal diagrams do not usually follow

consistent standards. They may exhibit inconsistencies, the lack of standardisation and clarity, varying levels of detail or abstraction, and the omission or inclusion of relevant or irrelevant system aspects and technologies. In combination, these aspects can hinder the understanding of MLOps system architecture descriptions.

There are various types of AI models, each serving unique purposes. ML models, for example, employ a range of algorithms to analyse data and make predictions. In this field, DL has emerged as a powerful approach, leveraging neural networks to tackle complex problems. RL plays a crucial role in addressing challenges in sequential decision-making and advancing automation and intelligence [44]. Recently, transfer learning has gained popularity in both DL [45], [46] and RL [47], [48], [49] as an effective strategy for leveraging knowledge across different domains. This approach enhances a model's adaptability and effectiveness in various tasks.

In DL and RL systems, the source code often serves as the primary artefact for understanding the ML workflow and implementing best practices [44], [50], [51], [52]. However, this code can be intricate and challenging to navigate, particularly for newcomers to the field. This complexity poses challenges for novice developers, especially as more software developers without formal AI training are required to engage with these systems. Meanwhile, data scientists and AI engineers who lack a strong background in software engineering often find themselves responsible for software engineering tasks within AI workflows.

Research Problem 3 (RP_3)

RP_3 *There is a lack of modelling approaches and tools for MLOps, DL and RL-based systems so that they are understandable, can be reasoned about, and programmatically analysed.*

ML systems introduce distinct challenges that differ from traditional software architectures, primarily due to the complex interactions between software development, data science, data engineering and the deployment of ML models as services, alongside MLOps practices that share similarities with DevOps. In the realm of MLOps and RLOps, automation plays a crucial role, facilitating the deployment of models and contributing to key architectural quality attributes. This emphasis on automation becomes particularly important in scenarios where model deployments involve continuous delivery.

The structure of ML applications faces several challenges, including intricate dependencies between software modules [53], technical debt and anti-patterns [38]. Additionally, there are design hurdles in ML model engineering, such as the selection and reuse of models [54], [55]. Factors such as model provisioning and monitoring ML service performance in distributed environments add to this complexity and highlight the difficulties in organising and decomposing ML systems. The diverse range of practices [23], [24], [56] complicates the identification of the ADDs that have been implemented. The variety of programming languages and supporting technologies further inhibits the programmatic analysis of systems for ADDs, especially when it comes to source code parsing. It is equally difficult to pinpoint the decisions architects made, the specific patterns and practices adopted and the degree of support for fundamental architectural quality attributes,

particularly automation, which is vital for MLOps.

In RL, developing agents that can make well-informed decisions is crucial. As AI systems grow in complexity, the need for standardised, automated methods for conformance checking becomes clear. In RL, the success of an agent in reaching its objectives strongly depends on the training strategies [2] applied. One challenge in implementing RL is determining which specific training methods are used, which is essential for evaluating adherence to best practices.

Research Problem 4 (RP_4)

RP_4 *Ensuring conformance to ADDs, patterns, practices, and decision options in MLOps and RLOps systems is a laborious and error-prone manual task. Objective measures, like metrics, for assessing the ADD options are also lacking. As ML and RL-based systems become increasingly complex, there is a need for standardised, automated detection of conformance to best practices, particularly regarding automation and training methods.*

MLOps workflows play a crucial role in ensuring reproducibility and reducing risks linked to model management and deployment [31], [57]. Although advancements have been made in MLOps, integrating ML models into production systems remains challenging. Traditional approaches typically require substantial manual effort to transition ML scripts from experimental settings, such as computational notebooks in Google Colab or Project Jupyter, to production-ready delivery pipelines [58]. This manual transition leads to inefficiencies and increases the potential for errors, hindering scalability and rapid deployment [59]. As a result, there is a pressing need for methodologies that can automate and streamline the development of production pipelines for MLOps, thereby reducing the dependence on extensive, specialised DevOps and software engineering expertise [31], [60].

In Industry 4.0 environments, where digital and physical technologies are integrated, CPPSs are increasingly leveraging RL models to enhance and automate production processes through RLOps. However, manually configuring RLOps pipelines, as in MLOps, requires specialised DevOps expertise, which can limit the overall automation potential. Within CPPSs, RL can be employed to train intelligent agents with digital twins [16], [61]. Therefore, the distinction between RLOps in Industry 4.0 and the functionality of cloud-based ML systems is stark. Unlike homogeneous cloud environments that are centrally controlled, an Industry 4.0 CPPS imposes unique constraints on the design and deployment of pipelines. Automation is highly valued in MLOps, RLOps and Industry 4.0 CPPSs. However, a critical component of automation – developing CI/CD pipelines – remains primarily manual. Pipeline development requires substantial effort and specialised knowledge in DevOps, software engineering and ML/RL [31], [60]. The manual nature of this task can result in scalability issues in the rapidly evolving context of Industry 4.0, which demands numerous swift and precise modifications to CI/CD pipelines across various physical deployment locations [59]. Consequently, there is a pressing need to automate the generation of RLOps CI/CD pipelines to address these challenges.

Research Problem 5 (RP_5)

RP_5 *Automating the ML/RL workflow requires specialist expertise and considerable manual effort to write and maintain pipeline configurations in an error-prone process that hinders scalability and rapid deployment.*

This thesis describes the work undertaken to better support practitioners by holistically unifying the key concepts described in Section 1.1 and addressing the research problems described in this section. A systematic alignment of technical decisions and operational strategies can be achieved by utilising ADDs as a basis for the architectural evolution process for MLOps [62]. Involving LLMs in the ADD generation process can increase its efficiency by automating various aspects of architectural synthesis [63]. Such strengths are particularly impactful when applied to RLOps, as RL requires a flexible and resilient architecture. In the context of Industry 4.0 and CPPSs, these strengths can be leveraged to rapidly architect intelligent, interconnected manufacturing environments, continuously advancing operational excellence and innovation in complex digital-physical ecosystems [64].

1.4 Research Questions

To tackle research problems **RP_1**-**RP_5** defined in Section 1.3, we formulated five corresponding overarching research questions **RQ_1**-**RQ_5**, each of which comprises several subsidiary, more detailed research questions. The overarching research questions are defined as follows:

Research Question 1 (RQ_1)

What ADDs, options, decision drivers and relations are relevant for the ML workflow and deployment when productionising ML models?

$RQ_{1.1}$ *Which ADDs are available to choose from in the context of the ML workflow, and what are their corresponding decision options?*

$RQ_{1.2}$ *What are the relations between these decisions and their decision options?*

$RQ_{1.3}$ *Which decision drivers (forces) are relevant to the decision options?*

$RQ_{1.4}$ *Which ADDs are available to choose from in the context of deployment for ML, and what are their corresponding decision options?*

$RQ_{1.5}$ *What are the relations between these decisions and their decision options?*

$RQ_{1.6}$ *Which decision drivers (forces) are relevant to the decision options?*

Research Question 2 (RQ_2)

To what extent do MLOps practices and ADDs apply to training strategies and architectures in RL systems, particularly within CPPSs?

$RQ_{2.1}$ *Which patterns and practices do practitioners currently use for training strategies in RL architectures?*

$RQ_{2.2}$ *What are the relations between existing training patterns and practices? Specifically, which ADDs are pertinent when designing RL systems?*

$RQ_{2.3}$ *What are the influencing factors (i.e. decision drivers) in architecting RL systems in the eyes of the practitioner today?*

$RQ_{2.4}$ *To what extent are known ML and RL ADDs applicable to RL in a CPPS context?*

$RQ_{2.5}$ *Which ADD decision options typically used in ML and RL apply directly to a real-world CPPS that utilises RL?*

$RQ_{2.6}$ *Is it necessary to adapt ADD decision options typically used in ML for use with RL in a CPPS context?*

Research Question 3 (RQ_3)

To what extent do semi-formal MLOps, DL and RL architecture models and diagrams enhance the understandability of MLOps, DL and RL-based systems for developers?

$RQ_{3.1}$ *How does the exclusion or inclusion of semi-formal MLOps system diagrams affect the accuracy with which participants assess their performance in the context of their perceived task correctness?*

$RQ_{3.2}$ *To what extent does the provision of semi-formal architecture models, in addition to system source code, improve the understandability of DL and RL patterns and practices?*

Research Question 4 (RQ_4)

How can conformance to MLOps and RL ADDs and best practices - particularly automation support and RL training methods - be objectively assessed and automated?

$RQ_{4.1}$ *How can MLOps systems' support for the core quality aspect of automation be assessed in the context of ADDs and their respective decision options?*

$RQ_{4.2}$ *What types of metrics can be applied to evaluate these levels of support, and how effective are they?*

$RQ_{4.3}$ *How effectively can LLMs assess best practices in RL training pipelines compared to heuristic-based code detectors?*

$RQ_{4.4}$ *What are the key architectural rules required for compliance with the best practices for checkpoints, hyperparameter tuning, training/agent configuration and single vs multi-agent RL in RL-based systems?*

Research Question 5 (RQ_5)

How accurately can LLM-powered low-code approaches generate and validate correct MLOps/RLOps pipeline configurations for RL workflows, particularly in Industry 4.0 contexts?

$RQ_{5.1}$ *How accurately can a low-code, template-based modular flow approach – leveraging LLMs for generation and validation – produce correct MLOps pipeline configurations for RL?*

$RQ_{5.2}$ *How accurately do different LLMs generate MLOps pipeline configurations for RL, and what are their limitations?*

$RQ_{5.3}$ *How accurately can a low-code, template-based, modular flow approach using LLMs generate Industry 4.0-specific RLOps pipeline configurations?*

$RQ_{5.4}$ *How do popular LLMs perform in generating Industry 4.0-specific RLOps pipeline configurations, and do they commit or encounter any recurring errors or challenges?*

1.5 Research Contributions

To answer research questions **RQ_1** - **RQ_5** from Section 1.4, we provide novel knowledge, methods and tools in the form of 10 research contributions **RC_1**-**RC_{10}**, described below. Contributions **RC_1** and **RC_2** address **RQ_1**, **RC_3** and **RC_4** address **RQ_2**, **RC_5** and **RC_6** address **RQ_3**, **RC_7** and **RC_8** address **RQ_4**, and **RC_9** and **RC_{10}** address **RQ_5**. Together, these 10 research contributions form our overarching approach, which is described in Section 1.9.

Research Contribution 1 (RC_1)

RC_1 *We systematically studied architectural knowledge by conducting a Straussian Grounded Theory [65] study of practitioner (grey) literature for the ML workflow. We catalogued 10 core ADDs together with 43 decision options and 30 decision drivers, providing design guidance for practitioners, as existing practitioner guidelines and best practices are often informal and inconsistent. We developed a novel ADD model and established reusable architectural knowledge in the form of ADDs, which form the foundation of our overarching approach and help bridge the gap between science and practice. Our formal ADD Python models are visualised in automatically generated Unified Modeling Language [66] (UML)-based model diagrams.*

Research Contribution 2 (RC_2)

RC_2 *We systematically studied deployment architectural knowledge by conducting a Straussian Grounded Theory [65] study of practitioner (grey) literature for ML deployment. We catalogued seven ADDs, 26 decision options, 44 decision drivers, and various relations in a Python-based ADD model, visualised in UML, providing design guidance for practitioners facing technical challenges due to knowledge gaps and interdisciplinary misalignments. Our results bridge the gap between science and practice, increase understanding of practitioner approaches, and support decision-making in deploying ML solutions to production.*

Research Contribution 3 (RC_3)

RC_3 *We conducted a qualitative, in-depth study of RL training strategies by applying a model-based qualitative research method to capture practitioners' best practices, patterns, and decision rationales. We systematically analysed 33 knowledge sources and distilled recurring relationships and decision drivers that shape training-strategy choices and their architectural consequences. Based on this evidence, we introduced a formal ADD model that encapsulates six decisions, 29 decision options, and 19 decision drivers to bridge scientific insights and practical RL architecture implementation and to provide structured decision-making support.*

Research Contribution 4 (RC_4)

RC_4 *We conducted an exploratory, qualitative, deductive-inductive industry case study on an Industry 4.0 CPPS and performed content analysis [67] of CPPS artefacts like architectural schematics and source code to understand their relation to known ADDs and associated decision options. In doing so, we fostered an initial understanding of RLOps architectures and provided a reference for practitioners that can be used as design guidance.*

Research Contribution 5 (RC_5)

RC_5 *We conducted a controlled experiment with 63 participants investigating the understandability of MLOps system architecture descriptions based on informal textual and graphical representations compared to semi-formal MLOps system diagrams. Results indicate that understandability – quantified by task correctness – is significantly greater using supplementary semi-formal diagrams, without significantly increasing task duration, and that task correctness correlates significantly with duration only when semi-formal diagrams are provided.*

Research Contribution 6 (RC_6)

RC_6 *In a controlled experiment with 158 participants, we assessed the understandability of ML-based systems and workflows through source code inspection versus semi-formal model representations. Findings show that providing semi-formal system diagrams with source code improves effectiveness for DL-relevant tasks, whilst RL-relevant tasks favoured source code alone; task durations increased slightly but showed no significant differences. These results demonstrate that semi-formal diagrams modelling workflow details like transfer learning and checkpoints benefit specific scenarios in understanding DL and RL practices.*

Research Contribution 7 (RC_7)

RC_7 *We developed novel, technology-agnostic metrics that offer reliable insights into support for automation – a fundamental pillar and architectural quality aspect – in MLOps systems, especially for continuous delivery of ML models. Validated by ordinal regression analysis and aligned with typical ADDs for automation in MLOps, our method enables systematic evaluation of automation-related ADDs and decision options through automated processes within CI/CD pipelines. It can be modified and extended to evaluate any relevant architectural quality aspects, thereby enhancing compliance with non-functional requirements and streamlining development, quality assurance and release cycles.*

Research Contribution 8 (RC_8)

RC_8 *We implemented a rule-based framework that integrates LLMs and heuristic-based code detectors to assess compliance with best practices in RL training pipelines [68]. Our architectural rules focus on best practices for RL-based architectures that are particularly relevant to CPPSs, including checkpoints, hyperparameter tuning, and agent configuration. We validated our solution via open-source projects and an Industry 4.0 CPPS case study.*

Research Contribution 9 (RC_9)

RC_9 *We developed a novel solution for generating MLOps pipelines for RL: a low-code, template-based approach that leverages LLMs and a human-in-the-loop to automate pipeline generation, validation and deployment. We evaluated our solution against seven LLMs and three open source projects. Pipelines generated by our approach had low error rates, and after a single round of improvements to the implementation, resulted in no errors for the best-performing LLM. This solution can potentially enable rapid scaling and deployment of reliable MLOps for RL pipelines, particularly for practitioners lacking advanced software engineering or DevOps skills.*

Research Contribution 10 (RC_{10})

RC_{10} *We applied our refined approach from **RC_9** to an Industry 4.0 CPPS case study in collaboration with our industry partner, Siemens Aktiengesellschaft Österreich, and achieved perfect results in our evaluation.*

1.6 Publications

Each of the 10 research contributions **RC_1**-**RC_{10}** described in Section 1.5 is associated with a corresponding publication **P_1**-**P_{10}** below, each of which has been published in a peer-reviewed scientific journal or at a peer-reviewed conference, and forms the basis of a chapter of the thesis. For each topic covered by a publication, the respective chapter discusses related work and provides details as appropriate, outlines limitations and identifies threats to validity. We also reference replication packages in the corresponding chapters.

Publication 1 (P_1)

P_1 S. J. Warnett and U. Zdun, "Architectural Design Decisions for the Machine Learning Workflow", *Computer*, vol. 55, no. 3, pp. 40–51, 2022. DOI: `https://doi.org/10.1109/MC.2021.3134800` [23] (Chapter 2)

Publication 2 (P_2)

P_2 S. J. Warnett and U. Zdun, "Architectural Design Decisions for Machine Learning Deployment", in *2022 IEEE 19th International Conference on Software Architecture (ICSA)*, IEEE, 2022, pp. 90–100. DOI: `https://doi.org/10.1109/ICSA53651.2022.00017` [24] (Chapter 3)

Publication 3 (P_3)

P_3 E. Ntentos, S. J. Warnett and U. Zdun, "Supporting Architectural Decision Making on Training Strategies in Reinforcement Learning Architectures", in *2024 IEEE 21st International Conference on Software Architecture (ICSA)*, 2024, pp. 90–100. DOI: 10.1109/ICSA59870.2024.00017 [8] (Chapter 4)

Publication 4 (P_4)

P_4 S. J. Warnett and U. Zdun, "Bridging the Gap Between MLOps and RLOps: An Industry 4.0 Case Study on Architectural Design Decisions in Practice", in *2025 IEEE 22nd International Conference on Software Architecture (ICSA)*, 2025, pp. 232–242. DOI: 10.1109/ICSA65012.2025.00031 [9] (Chapter 5)

Publication 5 (P_5)

P_5 S. J. Warnett and U. Zdun, "On the Understandability of MLOps System Architectures", *IEEE Transactions on Software Engineering*, vol. 50, no. 5, pp. 1015–1039, 2024. DOI: 10.1109/TSE.2024.3367488 [69] (Chapter 6)

Publication 6 (P_6)

P_6 E. Ntentos, S. J. Warnett and U. Zdun, "On the Understandability of Machine Learning Practices in Deep Learning and Reinforcement Learning Based Systems", *Journal of Systems and Software*, vol. 222, p. 112 343, 2025, ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2025.112343. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121225000111 [70] (Chapter 7)

Publication 7 (P_7)

P_7 S. J. Warnett, E. Ntentos and U. Zdun, "A Model-Driven, Metrics-Based Approach to Assessing Support for Quality Aspects in MLOps System Architectures", *Journal of Systems and Software*, vol. 220, p. 112 257, 2025, ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2024.112257. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121224003017 [71] (Chapter 8)

Publication 8 (P_8)

P_8 E. Ntentos, S. J. Warnett and U. Zdun, "Rule-Based Assessment of Reinforcement Learning Practices Using Large Language Models", in *2025 IEEE/ACM 4th International Conference on AI Engineering – Software Engineering for AI (CAIN)*, 2025, pp. 1–11. DOI: 10.1109/CAIN66642.2025.00009 [68] (Chapter 9)

Publication 9 (P_9)

P_9 S. J. Warnett, E. Ntentos and U. Zdun, "MLOps Pipeline Generation for Reinforcement Learning: A Low-Code Approach Using Large Language Models", *Journal of Systems and Software*, vol. 235, p. 112 760, 2026, ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2025.112760. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121225004297 [72] (Chapter 10)

Publication 10 (P_{10})

P_{10} S. J. Warnett, U. Zdun and S. Geiger, "RLOps Pipeline Development with Low-Code and Large Language Models for Industry 4.0", in *CAIN 2026 - 5th International Conference on AI Engineering - Software Engineering for AI*, Jan. 2026. DOI: https://doi.org/10.1145/3793653.3793779. [Online]. Available: http://eprints.cs.univie.ac.at/8676/ [73] (Chapter 11)

Publication P_2 won the Best Paper Award at the IEEE International Conference on Software Architecture (ICSA) 2022. On this basis, we presented it at the IEEE/ACM International Conference on Software Engineering (ICSE) 2023 at the organisers' invitation. The journal publications P_4 and P_7 were additionally accepted as journal-first submissions and presented at the IEEE/ACM International Conference on Software Engineering (ICSE) 2025 and the European Conference on Software Architecture (ECSA) 2025, respectively.

Two further publications deserve mention. Publication P_{11} partially forms the basis for some content in Chapters 1 and 12. Publication P_{12} is related to the thesis in its methodology and domain. The methodological relevance lies in applying the same practitioner-centred, model-based research approach that aims to support architectural decision-making to develop a formal ADD model. This approach is similar to that applied in publications P_1-P_3. Similarities in its domain pertain to its application to cyber-physical systems (as in the case studies of publications P_4, P_8 and P_{10}). It differs in that its focus is on data integration in distributed cyber-physical systems rather than on ML/RL-specific aspects. Since it does not entirely fit within the scope of this thesis, its content has been omitted.

Publication 11 (P_{11})

P_{11} S. J. Warnett and U. Zdun, "AI-Powered Architecting for Industry 4.0 Cyber-Physical Production Systems: A Novel Approach, Research Problems and Challenges", in *Software Architecture. ECSA 2025 Tracks and Workshops*, Cham: Springer Nature Switzerland, 2026, pp. 155–168, ISBN: 978-3-032-04403-7. DOI: https://doi.org/10.1007/978-3-032-04403-7_15 [74]

Publication 12 (P_{12})

P_{12} E. Ntentos, A. Amiri, S. Warnett and U. Zdun, "Decision-Making Support for Data Integration in Cyber-Physical-System Architectures", in *Service-Oriented Computing*, Cham: Springer Nature Switzerland, 2023, pp. 137–152, ISBN: 978-3-031-48421-6. DOI: https://doi.org/10.1007/978-3-031-48421-6_10 [75]

Publications P_1-P_{12} were supported by the following funding agencies:

- FFG (Austrian Research Promotion Agency)
  - Project AMMONIS (no. 879705): publications P_1, P_2 and P_5.
  - Project MODIS (no. FO999895431): publications P_3-P_{12}.
  - Project BEAM (no. FO999933314): publication P_{10}.
- FWF (Austrian Science Fund)
  - Project CQ4CD (Grant-DOI: 10.55776/I6510): publications P_9-P_{11}.

1.7 Research Methods

The work described in this thesis addresses the research problems outlined in Section 1.3 using various research methods. This section summarises the research methods, which are detailed in later chapters.

Grounded Theory Our work on ADDs for the ML workflow and deployment, and best practices and patterns within training strategies for RL architectures (P_1, P_2 and P_3, respectively) involved the qualitative analysis of grey (practitioner) literature and uses Straussian Grounded Theory [76] (P_1 and P_2), and Grounded Theory methods together with pattern mining (P_3). We modelled current ML practices using our Python [77]-based modelling tool, Codeable Models [78], resulting in an ADD model grounded in current practitioners' understanding of the domain. In P_8, we also applied Grounded Theory coding practices to practitioner literature when developing heuristics for rule-based assessment of RL practices using LLMs.

Grounded Theory was particularly suitable for P_1-P_3 because their research questions were exploratory rather than hypothesis-driven and focused on predefined constructs. Using Grounded Theory helped us devise conceptual categories that explain practitioners' needs and options, the relations between these categories and an emergent theoretical model of the design space, through inductive theory-building from systematically analysed qualitative data in an under-theorised domain. The resulting ADDs, decision options, decision drivers and relations codify current ML and RL practices.

Model-Based Research As mentioned above, model-based qualitative research was applied in P_1-P_3, leading to the development of a formal ADD metamodel, the implementation of model instances in Python and the generation of semi-formal UML

representations of our ADD models. The introduction of a formal ADD model helps to bridge the gap between scientific knowledge of the domain and practitioner understanding. Thus, we systematically integrated Grounded Theory techniques (coding and constant comparison) with metamodel-driven UML modelling to produce reusable ADD artefacts. We also use models as part of **P4** when conducting an industry case study to model part of Siemens's system. In **P5** and **P6**, we modelled MLOps, DL and RL-based systems for controlled experiments and in the model-driven, metrics-based empirical validation of **P7**, we also conducted modelling for performing the automated conformance checking of quality attributes.

Content Analysis In **P4**, we applied content analysis per DeFranco and Laplante [67], analysing existing Siemens project artefacts, including source code, CI/CD configurations, schematics, UML diagrams, presentations and commit histories, systematically coding artefacts against 22 predefined ADDs and 86 decision options from our prior Grounded Theory work. This process was deductive-inductive, comprising deductive classification into ADDs and inductive emergence of categories from findings. We combined content analysis with modelling for validation and representation – relevant project artefacts were modelled using Codeable Models [78] and UML diagrams for components and processes were generated with PlantUML [79]. This approach aided transparency and traceability, as well as triangulation and validation via formalisation.

Design Science According to Hevner et al. [80], "*in the design-science paradigm, knowledge and understanding of a problem domain and its solution are achieved in the building and application of the designed artifact*", and they further define artefacts as *constructs* (vocabulary and symbols), *models* (abstractions and representations), *methods* (algorithms and practices) and *instantiations* (implemented and prototype systems).

The ADD models described above, developed as part of **P1**-**P3**, can thus be considered design science artefacts. Metamodels and model instances of MLOps and RL systems were also developed and used in our industry case study in **P4**, and in **P5** and **P6**, where we conducted quantitative empirical controlled experiments (see below) and **P7**, where we performed a quantitative analysis of the properties of modelled systems, using model detectors to calculate metrics. Further design science artefacts are found in **P8**, which uses heuristic-based code detectors, and **P9** and **P10**, which use a tool we developed to generate CI/CD pipelines for MLOps and RLOps based on low-code specifications. Three of our studies (**P8**-**P10**) make use of LLMs, extending our prior work on ADDs, MLOps and RLOps architectures by integrating LLMs into rule-based assessment and low-code pipeline generation.

We employed design science methodologies with empirical validation through open-source and industry case studies, using controlled evaluations and metrics such as precision, recall, F_1 score and error rates. Design science and case-based empirical validation were particularly suited to these studies because the research questions were centred on building and demonstrating the feasibility of LLM-enhanced tools for RLOps. Easterbrook et al. [81] acknowledge that "*it is often necessary to use a combination of methods to fully*

understand the problem". In this case, we used design science for artefact creation, case studies for contextual realism and metrics for objective, empirical quantitative evaluation.

Controlled Experiments When evaluating our system models for understandability, conducting controlled experiments enabled the collection of quantitative data and generalisation of the findings to broader populations. Other techniques, such as observational studies, eye tracking, surveys, comparative studies, qualitative interviews and usability studies, were deemed less suitable due to limitations in establishing causality and controlling biases. Controlled experiments, on the other hand, provide necessary control over variables and ensure the validity of the results. We deliberately chose a between-subject experimental design in $\mathbf{P_5}$ and a within-subject design in $\mathbf{P_6}$ [82].

When planning our studies, we followed the template provided by Jedlitschka et al. [83], which details robust methods for empirical research in software engineering. We also adhered to guidelines according to Kitchenham, Pfleeger et al. [84], who offer overarching guidelines for conducting software engineering experiments and provide recommendations related to the design, implementation, analysis and presentation of empirical research studies. We also followed Wohlin et al. [85], who go into more detail and discuss statistical tests and their suitability for various study types, and Juristo and Moreno [86], evaluating the data from the experiment sessions using the robust statistical methods guidelines for empirical software engineering by Kitchenham, Madeyski et al.[87].

Industry Case Study $\mathbf{P_4}$ employs an exploratory, observational qualitative industry case study methodology on our industry partner's CPPS, adhering to the case study guidelines outlined by Runeson and Höst [88], who state that "*case study is a suitable research methodology for software engineering research since it studies contemporary phenomena in its natural context*" and Yin [89], who describes a case study as an empirical method that "*investigates a contemporary phenomenon (the 'case') in depth and within its real-world context, especially when the boundaries between phenomenon and context may not be clearly evident*". As mentioned previously $\mathbf{P_8}$ and $\mathbf{P_{10}}$ also use industry case studies, but quantitative ones rather than qualitative ones.

1.8 Research Overview

Table 1.1 provides an at-a-glance overview of the research process detailed in this thesis. It links research problems, research questions, research contributions, research methods, publications and thesis chapters. For example, $\mathbf{RP_1}$ is associated with $\mathbf{RQ_1}$, $\mathbf{RC_1}$, and $\mathbf{RC_2}$. $\mathbf{RC_1}$ is associated with $\mathbf{P_1}$, which applies the research methods Grounded Theory, modelling and design science and is described in Chapter 2. $\mathbf{RC_2}$ is associated with $\mathbf{P_2}$. $\mathbf{P_2}$ also applies the research methods Grounded Theory, modelling and design science, and is described in Chapter 3.

Table 1.1: Research Overview at a Glance

Research Problems	Research Questions	Research Contributions	Publications	Research Methods	Chapters
RP$_1$ (ML knowledge gaps)	**RQ$_1$** (ADDs for ML)	**RC$_1$** (ML workflow ADDs catalogue & model)	**P$_1$**	Grounded Theory Modelling Design Science	2
		RC$_2$ (ML deployment ADDs catalogue & model)	**P$_2$**	Grounded Theory Modelling Design Science	3
RP$_2$ (MLOps-RL fit unclear)	**RQ$_2$** (ADDs for RL/CPPSs)	**RC$_3$** (RL training ADDs catalogue & model)	**P$_3$**	Grounded Theory Pattern Mining Modelling Design Science	4
		RC$_4$ (Industry 4.0 RLOps architecture analysis)	**P$_4$**	Case Study Modelling Design Science Content Analysis	5
RP$_3$ (No MLOps/DL/RL modelling methods for understanding)	**RQ$_3$** (MLOps models for understanding)	**RC$_5$** (Semi-formal MLOps diagrams improve understanding)	**P$_5$**	Design Science Modelling Experiments	6
		RC$_6$ (Semi-formal DL/RL diagrams improve understanding)	**P$_6$**	Design Science Modelling Experiments	7
RP$_4$ (Manual conformance checks are error-prone)	**RQ$_4$** (Automate MLOps/RLOps conformance checks)	**RC$_7$** (Conformance metrics for MLOps)	**P$_7$**	Design Science Modelling	8
		RC$_8$ (LLM RL conformance assessment method)	**P$_8$**	Grounded Theory Design Science Case Study	9
RP$_5$ (Workflow automation complexity)	**RQ$_5$** (Automate MLOps/RLOps pipeline generation)	**RC$_9$** (LLM low-code RL pipeline generation)	**P$_9$**	Design Science	10
		RC$_{10}$ (LLM low-code RLOps pipeline generation for Industry 4.0)	**P$_{10}$**	Design Science Case Study	11

1.9 Overarching Approach

The integration of AI into software architecture and engineering has the potential to greatly support practitioners in developing ML and RL systems. Integrating AI into software architecture can catalyse rapid advancements in how practitioners design, optimise and maintain systems [90]. AI-driven techniques transform traditional practices by documenting decision-making processes, refining architectural trade-offs and enabling dynamically adaptive systems capable of self-evolution [91], [92].

The role of AI in overcoming long-standing issues in software architecture, including complexity management, up-to-date documentation and the continuous evolution of

systems, is becoming increasingly important [90]. The manual expertise, effort and sophisticated reasoning required to apply traditional architecting practices cannot keep pace with the demanding architectural needs and operational uncertainty of AI-enabled systems [90]. Combined with the highly dynamic, rapidly changing and unpredictable nature of the domains in which architectures are applied, for instance in CPPSs [15], there is an urgent need to investigate how AI can automate the architectural decision-making process that caters for such domains [9], [90], [93] in ML [94] and RL-based systems.

We followed a systematic approach to understand and support practitioners in better designing, building and managing ML/RL systems in an automated manner, using AI where appropriate. Our approach has evolved, starting with asking fundamental questions about practitioners' understanding of architectural decisions and developing into the implementation of tools in an Industry 4.0 context for CPPSs that utilise LLMs, enabling practitioners to make better decisions and automate specific tasks. The work described towards the end of this thesis, in Chapters 9, 10 and 11, is an effort to apply AI-powered architecting to enhance automation and reduce manual effort. In Chapter 12, we describe how manual aspects of our overarching approach in supporting the architecting process could be enhanced using AI and four further key open research challenges.

Overview of Our Approach Figure 1.2 illustrates our approach, comprising the key contributions. The dashed arrowed lines indicate which works were prerequisites for others. Fundamental to the overall strategy was the initial consolidation of **Architectural Knowledge** into ADDs for ML and RL, along with an understanding of the emerging paradigm RLOps. We then used this knowledge to develop an **Architectural Modelling** method and tools for automating the assessment of **Architectural Conformance** using model detectors and LLMs. **Architectural Knowledge** is also applied to our **Pipeline Generation** technique, which uses LLMs in a template-based, low-code approach.

Our novel approach applies the 10 research contributions described in Section 1.5: **RC$_1$** workflow ADDs; **RC$_2$** ML deployment ADDs; **RC$_3$** RL training ADDs; **RC$_4$** extending MLOps into RLOps; **RC$_5$** introducing formal architectural models for MLOps; **RC$_6$** adding formal architectural models for DL/RL; **RC$_7$** adding automatic conformance checks with detectors; **RC$_8$** introducing LLMs in RL conformance assessment; **RC$_9$** creating MLOps pipelines for RL using LLMs and a low-code approach; **RC$_{10}$** extending the LLM pipeline creation to RLOps for Industry 4.0 CPPSs.

Placement of Our Approach in a CI/CD Pipeline Figure 1.3 shows how our overall approach fits into the context of CI/CD pipelines for ML and RL, particularly that of Siemens in the context of Industry 4.0 and CPPSs. The figure illustrates the high-level, essential steps in a CI/CD pipeline common to ML and RL, augmented with our contributions, which are coloured purple, and traditional CI/CD steps, coloured grey. Our research contributions act either as a basis for input to other research contributions or pipeline steps, shown by dotted line arrows, or as pipeline execution steps, linked with unbroken line arrows. The unbroken line arrows show the execution flow direction within the CI/CD pipeline, starting with the circle before the pipeline trigger step and

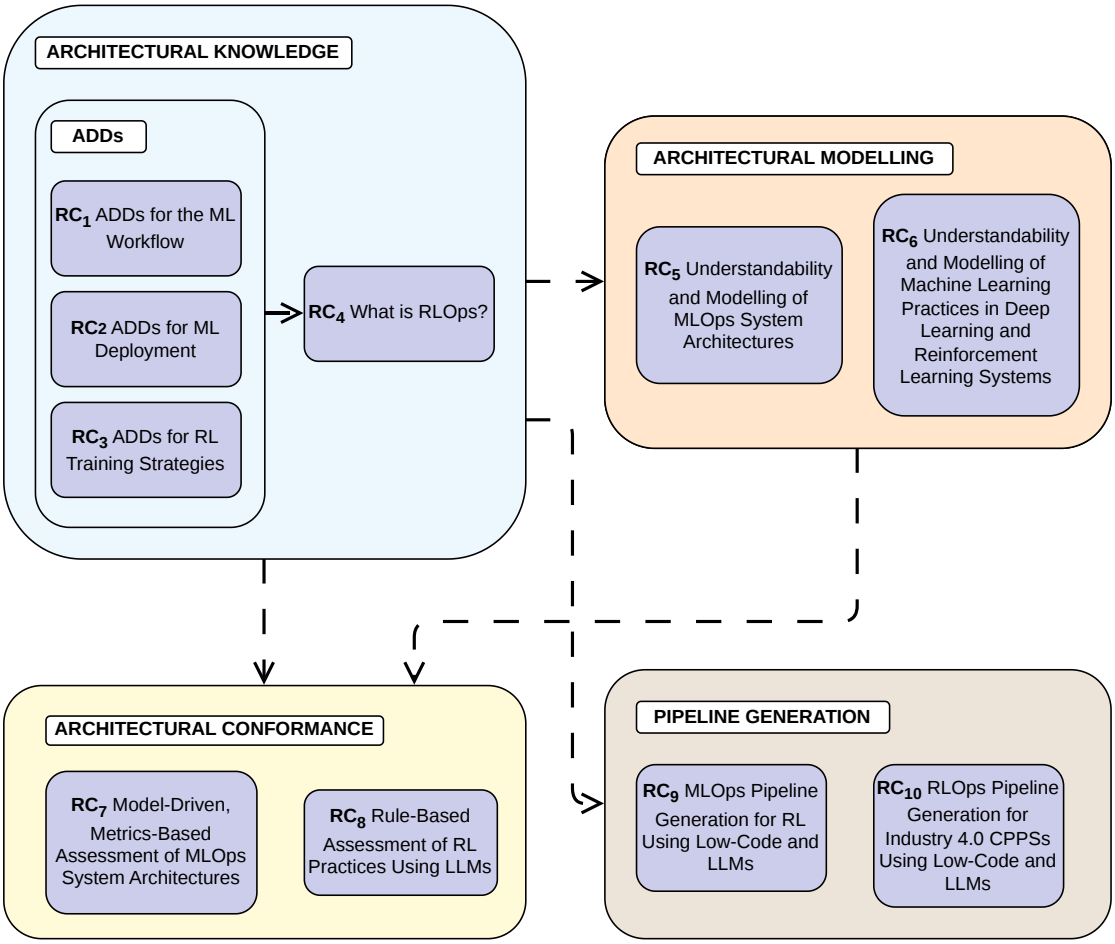

Figure 1.2: Overview of our approach.

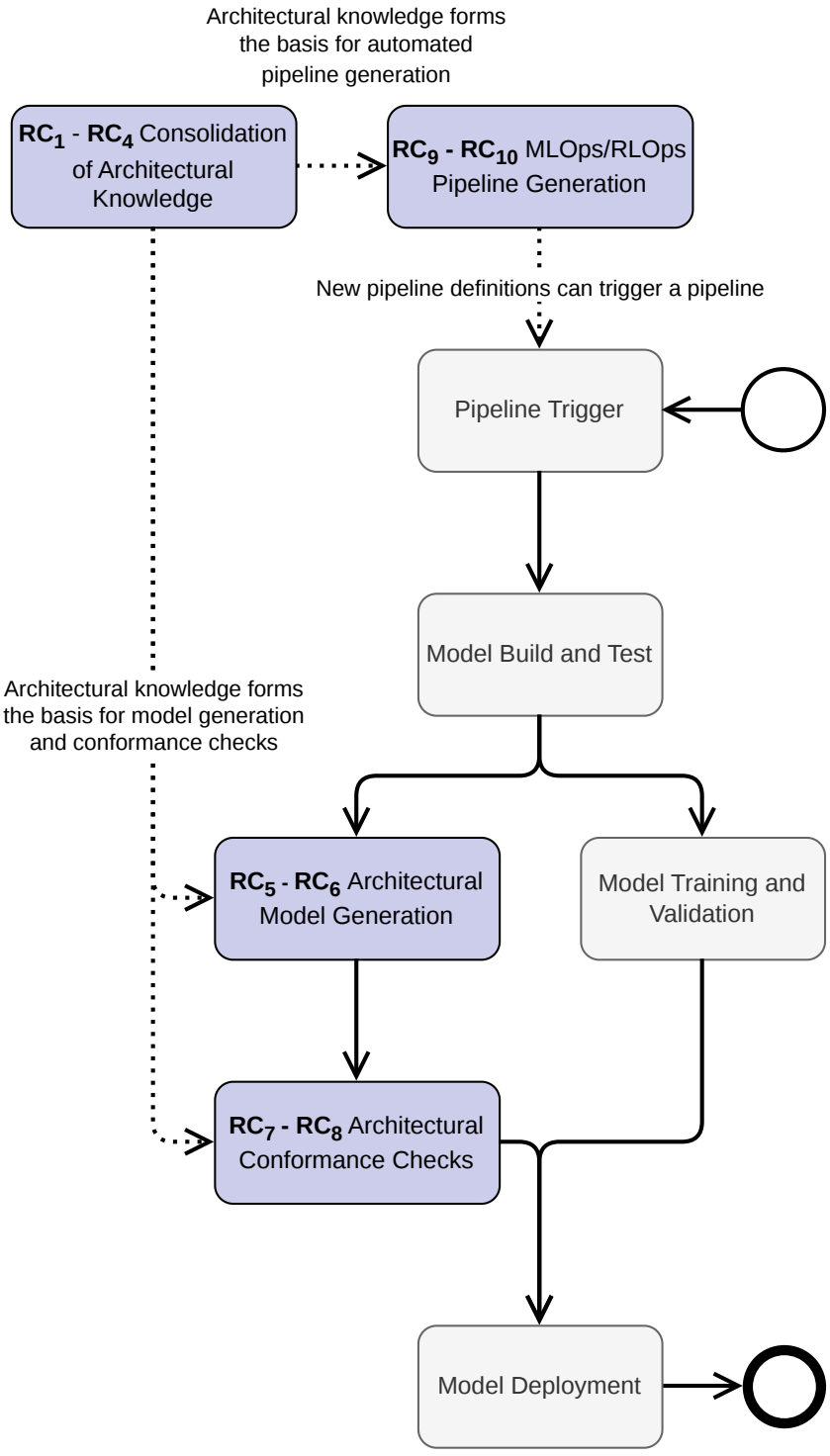

Figure 1.3: Placement of our approach in a CI/CD pipeline.

ending with the double circle after deployment.

The basis for the entire process is **RC$_1$**-**RC$_4$** **Consolidation of Architectural Knowledge**. It is an activity that only needs to be performed rarely, as new practices and patterns are unlikely to be documented in grey literature very frequently. Based on any new architectural knowledge, the components connected by dotted line arrows may be updated.

Initially, the pipeline is triggered in the usual way (**Pipeline Trigger**), such as when changes to the source code are made, or when an external component triggers it, for example, due to model performance drift or when the pipeline definition itself is updated. In an Industry 4.0 CPPS setting, where frequent changes are made to pipeline configurations and new pipelines need to be rapidly generated at scale, we can trigger a new pipeline with **RC$_9$**-**RC$_{10}$** **MLOps/RLOps Pipeline Generation**, in which our LLM-based low-code tool generates a new pipeline configuration and commits it, for instance, to a project's GitLab repository.

Following the **Model Build and Test** phase, both **Model Training and Validation** and **RC$_5$**-**RC$_6$** **Architectural Model Generation** can run in parallel. The latter phase uses **RC$_1$**-**RC$_4$** **Consolidation of Architectural Knowledge** as a basis. Currently, **RC$_5$**-**RC$_6$** **Architectural Model Generation** is a partly manual activity due to the distributed and polyglot nature of ML/RL-enabled systems, which makes it hard to create formal models from source code or architectural schematics using traditional methods such as source code parsing, and requires considerable expertise in a time-consuming process. Fortunately, the system models do not need to be updated often in practice – only when the architecture changes. With modern AI tools, we think system architecture model generation could be automated, and we discuss this further in Section 12.2.

The next step is **RC$_7$**-**RC$_8$** **Architectural Conformance Checks**, an automated task that checks architectural conformance against specific quality attributes, based on the provided system model, considering ADDs and associated practices and patterns. **RC$_7$** is largely manually programmed, but only needs to be updated if the system model, architectural knowledge or metrics change. Nevertheless, we envisage an enhancement using techniques similar to **RC$_8$**, combining LLMs with the consolidated **Architectural Knowledge**. We describe this proposed improved process in Section 12.2. **RC$_8$** runs on the RL source code and checks for specific architectural practices. Assuming the conformance checks, model training and validation succeeded, we proceed to the **Model Deployment** stage, at which point the pipeline can deploy the model, and the pipeline run terminates.

Part II

Architectural Knowledge for Machine Learning and MLOps

2 Architectural Design Decisions for the Machine Learning Workflow

Productionising ML models is a challenging process, as it is often fraught with uncertainty and confusion, partly because of the disparity between software engineering and ML practices and partly because of knowledge gaps at the individual practitioner level. We conducted a qualitative investigation into the architectural decisions faced by practitioners as documented in grey literature based on Straussian Grounded Theory, and modelled current practices in ML. Our novel ADD model is grounded in current practitioner understanding and helps bridge the gap between science and practice, fostering a deeper understanding of the subject and supporting practitioners by integrating and consolidating the myriad decisions they face. We describe a subset of the modelled ADDs, discuss uses for the model and outline areas for further research.

2.1 Introduction

Software engineering, software architecture and ML are established disciplines; however, a noticeable mismatch exists between engineering and architectural practices on the one hand and more data-focused activities on the other, despite the latter often involving complex software solutions. As a consequence, a rift has formed between the software engineering and ML communities [35]. This apparent discrepancy could be partially explained by acknowledging that ML developers are not typically from a software engineering background and thus do not routinely apply common engineering and architectural techniques. Conversely, software engineers and architects do not usually work as ML engineers and therefore do not regularly develop appropriate solutions for ML systems [36], [37]. The issues resulting from the divergence of these disciplines are further compounded by the technical challenges that are customarily encountered in ML projects but generally absent in non-ML projects, such as the development and maintenance of ML solutions [38], [39], [40].

To address these issues, we conducted an empirical grey literature study [95] based on Straussian Grounded Theory [76], [96]. We aimed to identify relevant architectural concepts applied by practitioners in ML data processing, model building and automated ML (AutoML). The result is a model of ADDs, decision options, practices, decision drivers and their relations.

In Section 2.2, we provide a brief description of a simplified ML pipeline workflow to illustrate our research context. We then describe our research questions and methodology in Section 2.3. Next, related studies in the field are discussed in Section 2.4.

After that, in Section 2.5, we present the ADDs resulting from our study. Finally, in Section 2.6, we conclude by briefly outlining future directions and potential research avenues based on our findings.

2.2 The ML Workflow

To illustrate the parts of ML architecture we consider in this chapter, Figure 2.1 depicts a typical ML workflow as a pipeline.

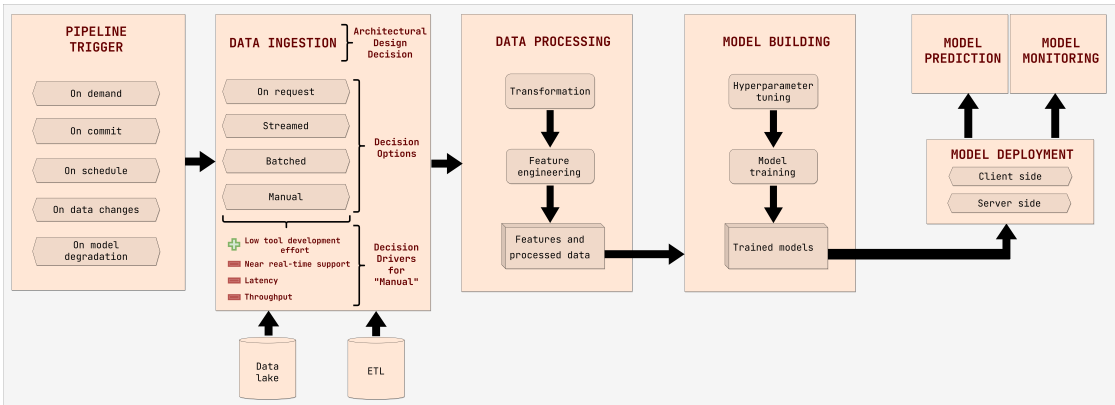

Figure 2.1: A simplified ML workflow as a pipeline.

An ML pipeline typically begins with a triggering event, which, for instance, may be initiated manually or following a codebase commit. This event initiates the data ingestion step, where the data sets are updated, e.g. by fetching the data afresh from a repository or using data received via a stream. The data is then processed in various ways and is typically transformed (e.g. normalised and then cleaned) before features are extracted. The features and processed data are then passed to the model-building components, in which models are trained. The trained models are then deployed and can subsequently be used for predictions.

2.3 Research Questions and Methodology

This chapter is based on $\mathbf{P_1}$, describes $\mathbf{RC_1}$ and addresses $\mathbf{RP_1}$ by aiming to answer the following research questions as part of $\mathbf{RQ_1}$:

$\mathbf{RQ_{1.1}}$ *Which ADDs are available to choose from in the context of the ML workflow, and what are their corresponding decision options?*

$\mathbf{RQ_{1.2}}$ *What are the relations between these decisions and their decision options?*

$\mathbf{RQ_{1.3}}$ *Which decision drivers (forces) are relevant to the decision options?*

To achieve this, we examined the methods and techniques currently used by practitioners in the context of ML solution development, gaining valuable insights into the state of the art in software engineering and architectural approaches for ML. The study was based on Straussian Grounded Theory [76], [96] and involved the qualitative analysis of grey literature [95], [97], which we have made use of in prior work outwith the scope of this thesis [98], [99]. We aimed to identify relevant ADDs, options, practices, decision drivers, and relations between them, whilst striving to reduce the gap between science and practice, and between software engineering and software architecture on the one hand, and ML on the other.

Grounded Theory Grounded Theory is a qualitative, inductive research method that links data analysis with theory. Iterative steps are taken when interpreting data, with the focus and central goal of building a theory grounded in the data. Data analysis should occur during data collection and not afterwards.

The most crucial activity in Grounded Theory is *constant comparison*: the researcher continuously and iteratively compares pre-existing data and concepts with new data. Any newly arising abstract concepts should then be compared with pre-existing concepts and data. The concepts are organised into categories (or *codes*) and are compared and linked to properties and each other via relations [65]. The concepts, categories and properties derived from the data should guide the next iteration of research activities.

Another central activity is *memo writing*, which documents the theory-building process, provides a basis for reflection, improves transparency by creating an audit trail and facilitates replicability. *Theoretical sampling* involves actively seeking out new data based on the results of the previous iteration, considering the kinds of data that should be collected next [100]. This process is continued until theoretical saturation is reached, i.e. "*the point in category development at which no new properties, dimensions, or relationships emerge during analysis*" [96].

We applied the methodology of Strauss and Corbin [65], which is characterised by three types of coding activity:

- *Open coding* involves developing concepts based on the data sources. It entails asking specific (and consistent) questions of the data, precise (and consistent) coding and memo writing with minimal assumptions.
- *Axial coding* is the development of categories and the linking of data, concepts, categories and properties.
- *Selective coding* refers to the integration of the categories that have been developed and their grouping around a central core category.

Grey Literature According to Garousi et al. [95], grey literature in software engineering can be defined as "*any material about software engineering that is not formally peer-reviewed nor formally published*", such as blog posts, articles, presentations and audio-video material [97]. Grey literature was chosen as the sole data source for this

study because we aimed to understand practitioners' views in this domain, and such sources are most representative of those views. This premise is confirmed by Rainer and Williams [101], who describe various benefits to grey literature sources in software engineering research, including that they "*provide information on practitioners' contemporary perspectives on important topics relevant to practice and to research*" and "*promote the voice of the practitioner*".

Methodology Figure 2.2 illustrates our research method. We searched for practitioner sources using standard search engines (e.g. Google, StartPage and DuckDuckGo) and topic portals (e.g. InfoQ and DZone). The initial source search term was "ML workflow". We then repeatedly and iteratively applied Grounded Theory coding practices and constant comparison to identify concepts, categories, properties and relations. The resulting entities were modelled in Python code, which was then used to generate UML visualisations of the models. The subsequent sources were then searched for using appropriate search terms based on topics identified in the previous iteration and guided by the research so far, with particular consideration given to topics needing coding and their potential contribution to the model. Practitioner articles were deemed candidates if they were relevant to the topic under consideration and did not appear to be marketing a business or product. Each author reviewed the other author's selection of sources for suitability.

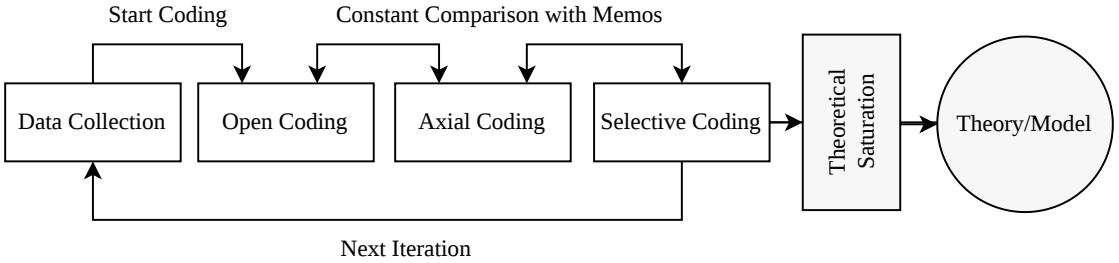

Figure 2.2: Our research method.

We applied open coding to transform conceptual details into conceptual labelling. Next, during axial coding, we identified categories based on recurring, synonymous and related concepts. During open and axial coding, we examined each source line individually. Our thought processes, conceptual understanding, interpretation and reasoning behind each coding decision were documented in memos for the various sources, facilitating the traceability of codes back to their sources.

Whilst conducting selective coding, we carved out the main ideas of the theory, gaining an understanding of the big picture by considering the data and analysis results. We also revisited and refined previous sources during the selective coding process. Finally, we employed formal Python-based modelling for axial and selective coding to develop a precise and consistent theory. Once around five to seven additional sources had been considered without adding further detail to our model, theoretical saturation was understood to have been reached. Our knowledge sources are summarised in Table 2.1. Twenty-nine sources were considered in total.

Table 2.1: List of Knowledge Sources Included in the Study

ID	Title	Archive URL	Source Type	Example	Source Code
s1	How to power up your product by machine learning with python microservice, pt. 1	https://tinyurl.com/ml-adds-u1	Practitioner Audience Article	True	False
s2	Architecting a Machine Learning Pipeline: How to build scalable Machine Learning systems – Part 2/2	https://tinyurl.com/ml-adds-u2	Practitioner Audience Article	True	False
s3	Some Thoughts on Modularization in Machine Learning	https://tinyurl.com/ml-adds-u3	Practitioner Audience Article	False	False
s4	Productionizing Machine Learning with a Microservices Architecture	https://tinyurl.com/ml-adds-u4	Presentation Video	False	False
s5	Microservices Suck for Machine Learning (and what we did about it)	https://tinyurl.com/ml-adds-u5	Practitioner Audience Article	True	True
s6	MLOps: Continuous delivery and automation pipelines in machine learning	https://tinyurl.com/ml-adds-u6	Practitioner Audience Article	True	False
s7	Composing Deep-Learning Microservices for the Hybrid Internet of Things	https://tinyurl.com/ml-adds-u7	Practitioner Audience Article	True	False
s8	Continuous Intelligence: Moving Machine Learning Application into Production Reliably	https://tinyurl.com/ml-adds-u8	Slides	True	False
s9	Architecture of a real-world Machine Learning system	https://tinyurl.com/ml-adds-u9	Practitioner Audience Article	True	False
s10	Architecting a Machine Learning System for Risk	https://tinyurl.com/ml-adds-u10	Practitioner Audience Article	True	False

Continued on next page...

ID	Title	Archive URL	Source Type	Example	Source Code
s11	Architecting a Scalable Real Time Learning System	https://tinyurl.com/ml-adds-u11	Practitioner Audience Article	True	False
s12	System Architectures for Personalization and Recommendation	https://tinyurl.com/ml-adds-u12	Practitioner Audience Article	True	False
s13	Architectural thinking in the Wild West of data science	https://tinyurl.com/ml-adds-u13	Practitioner Audience Article	True	False
s14	Machine Learning Architecture: The Core Components	https://tinyurl.com/ml-adds-u14	Practitioner Audience Article	False	False
s15	Scalable Software and Big Data Architecture – Big Data and Analytics Architectural Patterns	https://tinyurl.com/ml-adds-u15	Practitioner Audience Article	False	False
s16	Machine Learning in Production: Software Architecture	https://tinyurl.com/ml-adds-u16	Practitioner Audience Article	True	False
s17	AutoML	https://tinyurl.com/ml-adds-u17	Practitioner Audience Article	False	False
s18	AutoML is Overhyped	https://tinyurl.com/ml-adds-u18	Practitioner Audience Article	True	False
s19	Three Levels of ML Software	https://tinyurl.com/ml-adds-u19	Practitioner Audience Article	True	False
s20	MLOps: Methods and Tools of DevOps for Machine Learning	https://tinyurl.com/ml-adds-u20	Practitioner Audience Article	False	False
s21	MLOps: What It Is, Why it Matters, and How To Implement It (from a Data Scientist Perspective)	https://tinyurl.com/ml-adds-u21	Practitioner Audience Article	False	False
s22	MLOps Principles	https://tinyurl.com/ml-adds-u22	Practitioner Audience Article	False	False
s23	Machine Learning Monitoring: What It Is, and What We Are Missing	https://tinyurl.com/ml-adds-u23	Practitioner Audience Article	False	False

Continued on next page...

ID	Title	Archive URL	Source Type	Example	Source Code
s24	Automated monitoring of your machine learning models with Amazon SageMaker Model Monitor and sending predictions to human review workflows using Amazon A2I	https://tinyurl.com/ml-adds-u24	Blog Post	False	False
s25	MLOps: Model management, deployment, and monitoring with Azure Machine Learning	https://tinyurl.com/ml-adds-u25	Practitioner Audience Article	False	False
s26	The Pros and Cons of Using Jupyter Notebooks as Your Editor for Data Science Work TL;DR: PyCharm's probably better	https://tinyurl.com/ml-adds-u26	Practitioner Audience Article	False	False
s27	10 reasons why data scientists love Jupyter notebooks	https://tinyurl.com/ml-adds-u27	Practitioner Audience Article	False	False
s28	5 reasons why jupyter notebooks suck	https://tinyurl.com/ml-adds-u28	Practitioner Audience Article	False	False
s29	Jupyter Notebook is the Cancer of ML Engineering	https://tinyurl.com/ml-adds-u29	Practitioner Audience Article	False	False

Modelling was achieved using Codeable Models [78] – our Python tool for defining models and model instances in code and recording memos. Based on the Python code, PlantUML [79] code generators were used to generate UML-based graphical visualisations of the model[1], in addition to a textual model specification in Markdown[2] and Latex [102].

We describe only a subset of the ADDs modelled in this study – specifically, those central to the ML workflow. To support the replicability of our study and provide the full models, we offer detailed Grounded Theory coding and the resulting models, along with our Python implementation for generating UML diagrams and tables from the models, in our replication package [103].

[1] Generated UML figures have been optimised for space and readability.

[2] https://daringfireball.net/projects/markdown

2.4 Related Work

Some headway has already been made in the fields of software engineering and software architecture for ML. The systematic literature review conducted by Washizaki et al. [104] provides a comprehensive, categorised collection of patterns and anti-patterns for ML, and several other publications document the challenges and open research topics in the field.

Lwakatare et al. [39] conducted an empirical investigation of software engineering challenges in ML systems. They propose a taxonomy of issues that includes data dependency management, deployment difficulties and challenges in model and result replicability.

Sculley et al. [38] describe the significant technical debt incurred when attempting to maintain real-world ML systems. Such risk factors include entanglement, data dependencies, configuration issues, reproducibility debt and system-level anti-patterns.

Bosch et al. [105] provide an overview of the software engineering challenges associated with ML solutions, including model management, reuse and deployment, data pipeline challenges, monitoring and logging, design methods and data quality management. They also identify open research areas, such as distributed model creation, distributed data storage, data generation and automated experimentation.

Nascimento et al. [40] conducted a comprehensive systematic literature review on the subject of software engineering for AI and ML. They summarise the state of the art and identify several unsolved challenges in testing, AI software quality and data management. They also identify several other topics, such as architecture design, AI engineering, model development, model deployment, integration, infrastructure, operations support, requirements engineering, project management and education.

Serban et al. [106] mined academic and grey literature, identifying 29 engineering best practices for ML applications. They also surveyed practitioners to determine the degree of adoption of these practices and validate their perceived effects. Their findings helped improve the understanding of the relationship between team size and practice adoption, as well as the relative adoption rate of software engineering practices compared to ML-specific practices.

Martínez-Fernández et al. [107] conducted a systematic mapping study of 248 studies. They identified and classified various software engineering approaches for AI-based systems, highlighting prevalent and neglected areas of study.

Alves et al. [108] conducted a grey literature review to investigate techniques and practices applied to ML product lifecycles. They identified existing product engineering methods and practices for industrial ML applications and platforms.

In contrast, our approach was to formally model real-world practices, assisting practitioners in finding suitable options for common design decisions and mitigating challenges common to the field. Our ADDs can be used to assess the uncertainty or complexity of the ML system design space, evaluate architectural conformance or identify anti-patterns.

2.5 ADDs

Table 2.2[3] presents an overview of the various ADDs, the number of sources evidencing each ADD, the decision options, the grey literature sources and the decision drivers and their impacts. These are discussed in this section. Our replication package [103] includes detailed UML models of all ADDs and relations described in this chapter.

Table 2.2: Study Results: Overview of ADDs, Number of Sources, Decision Options, Sources and Related Decision Drivers

ADDs	#	**Decision Options**	**Practitioner Sources**	**Decision Drivers**
How to automatically process the data used for model building?	14	1. No data processing automation	s1, s6, s13	f1(--), f2(-), f3(-), f4(--), f5(+)
		2. Data pipeline	s1, s2, s4, s5, s6, s8, s9, s13, s14, s15, s16, s17, s18, s19	f1(++), f2(++), f3(+), f4(+), f5(-)
		3. ETL pipeline	s4, s5, s9, s13, s15	f1(+), f2(+), f3(o), f4(+), f5(-)
		4. Data processing component	s9, s13, s14	f1(+), f2(+), f3(-), f4(o), f5(-)
Which data processing tasks can be performed by a data processing pipeline or component?	19	1. Data extraction	s1, s2, s6, s8, s13, s14, s15	∅
		2. Data transformation	s2, s5, s6, s10, s13, s14, s15, s19	∅
		3. Data preparation	s1, s2, s4, s5, s6, s9, s10, s11, s13, s14, s15, s17, s18, s19, s20, s23, s24, s25	∅
		4. Data validation	s6, s9, s13, s19, s20, s24	∅
		5. Data selection	s2, s6, s14	∅
		6. Feature engineering	s1, s2, s4, s5, s6, s8, s9, s10, s13, s14, s15, s17, s18, s19, s23, s25	∅
		7. Data processing hyperparameter tuning	s9, s11, s17, s18	∅

Continued on next page...

[3] The following colour scheme in Table 2.2 indicates the source frequency:
□ < 5%, □ < 10%, □ < 20%, □ < 35%, □ < 50%, □ < 70%, □ ≥ 70%.

Design Decision	#	Decision Option	Practitioner Sources	Decision Drivers
How to persist and provide access to features?	9	1. Store features in data stores without specific support	s2, s4, s5, s6, s13	f6(-), f7(-), f8(-), f9(-), f10(o), f11(o), f12(o), f13(o), f14(o)
		2. Feature store	s2, s4, s5, s6, s9, s10, s11, s13, s15	f6(+), f7(+), f8(+), f9(+), f10(+), f11(+), f12(+), f13(+), f14(+)
Should data be processed in batches or in real time?	10	1. Batch-based data processing	s1, s2, s5, s6, s8, s10, s13, s15	f10(-), f11(+), f12(++), f13(-), f14(--), f15(++)
		2. Real-time, stream-based data processing	s1, s2, s4, s5, s6, s8, s9, s10, s13, s15	f10(++), f11(o), f12(+), f13(++), f14(++), f15(+)
How to ingest data into ML projects or applications?	11	1. Streamed data ingestion	s2, s4, s5, s7, s9, s11, s13, s14, s15, s19	f5(-), f10(+), f11(-), f12(+), f13(++), f14(++)
		2. Data ingestion on request	s2, s4, s5, s9, s13, s14, s15, s19	f5(-), f10(o), f11(+), f12(o), f13(o), f14(+)
		3. Data ingestion in batches	s4, s5, s13, s14, s15, s19	f5(-), f10(-), f11(o), f12(++), f13(-), f14(--)
		4. Manual data ingestion	s4, s5, s6	f5(+), f10(--), f11(-), f12(--), f13(--), f14(--)
How to trigger an ML pipeline or orchestrator?	9	1. On-demand trigger	s2, s4, s6, s9, s12	∅
		2. On commit trigger	s4, s8, s25	∅
		3. On-schedule trigger	s1, s2, s6, s9, s12, s14	∅
		4. On availability of new training data trigger	s6	∅
		5. On model performance degradation trigger	s6	∅
		6. On changes in the data distribution trigger	s6	∅
How to perform model building in an ML project?	13	1. Model building in development tool	s2, s4, s6, s10, s13, s15	f3(+), f4(-), f5(+), f13(--), f16(-), f17(+), f18(-), f19(--), f20(-), f21(-), f22(-), f23(-)
		2. Model-building pipeline	s2, s4, s6, s8, s9, s10, s13, s14, s17, s19, s25	f3(o), f4(+), f5(-), f13(+), f16(+), f17(-), f18(++), f19(+), f20(+), f21(+), f22(+), f23(+)
		3. Model-building component	s9, s11, s13, s14	f3(o), f4(o), f5(-), f13(+), f16(+), f17(-), f18(o), f19(o), f20(o), f21(+), f22(o), f23(-)

Continued on next page...

Design Decision	#	Decision Option	Practitioner Sources	Decision Drivers
Which tasks can be performed by a model-building pipeline or component?	21	1. Model training	s1, s2, s4, s5, s6, s7, s8, s9, s10, s11, s12, s13, s14, s15, s17, s18, s19, s20, s23, s24, s25	∅
		2. Data splitting	s2, s4, s8, s9, s14, s19	∅
		3. Data checkpoints	s2	∅
		4. Model validation	s4, s6, s8, s9, s10, s13, s14, s17, s18, s19, s20, s25	∅
		5. Model selection	s2, s4, s9, s11, s14, s17, s18, s19	∅
		6. Train multiple model versions with different parameters and/or algorithms	s2, s4, s6, s8, s9, s11, s19	∅
		7. Model packaging	s9, s19	∅
		8. Model hyperparameter tuning	s2, s6, s9, s11, s17, s18, s19, s24, s25	∅
		9. Development tool facade	s2, s4, s6	∅
		10. Development tool export	s10, s15	∅
When and how to train the model?	6	1. Batch-based learning	s2, s6, s10, s11, s12, s19	f10(-), f11(+), f14(-), f19(+), f24(--), f25(--), f26(++), f27(--), f28(++)
		2. Incremental learning	s6, s10, s11, s12, s19	f10(+), f11(-), f14(+), f19(-), f24(++), f25(++), f26(--), f27(+), f28(-)
		3. Hybrid batch-based and incremental learning	s12, s19	f10(+), f11(+), f14(o), f19(+), f24(+), f25(+), f26(+), f27(o), f28(--)
Should AutoML be used and, if so, where?	9	1. No AutoML	s4, s6, s9, s10, s17, s18, s19, s22, s23	f2(o), f4(o), f5(o), f20(+), f22(o), f29(o), f30(o)
		2. AutoML	s4, s6, s9, s10, s17, s18, s19, s22, s23	f2(+), f4(+), f5(o), f20(-), f22(+), f29(+), f30(+)

Continued on next page...

Forces Codes/Sources: **f1**: Automated data collection [s1], **f2**: Process and work automation [s1, s8, s15, s18, s20], **f3**: Iterative development [s1, s4], **f4**: Reproducibility [s6, s8, s13, s17, s20], **f5**: Tool development effort [s1, s2, s10, s17], **f6**: Feature reuse [s5, s6], **f7**: Feature discovery [s5, s6], **f8**: Maintainability [s1, s5, s6, s13, s16], **f9**: Avoid training/serving skew [s6, s8], **f10**: Online support [s2, s6, s9, s10, s11, s12, s13, s19], **f11**: Offline support [s2, s6, s9, s10, s11, s12, s13, s19], **f12**: Data throughput [s2, s4, s6, s9, s12, s15], **f13**: Low latency [s2, s4, s6, s15, s16], **f14**: Near real-time support [s1, s2, s4, s6, s10, s11, s12, s13, s15, s19], **f15**: Reliability [s2, s5, s8, s10], **f16**: Interchangeability of ML algorithms [s2], **f17**: Interactive prototyping [s2, s13], **f18**: Loose coupling [s2, s5, s6, s7, s16, s19], **f19**: Scalability [s1, s2, s3, s4, s5, s7, s11, s12, s13, s15], **f20**: Production-ready development [s4, s13, s16, s18, s19], **f21**: Experimental operational symmetry [s6, s13], **f22**: Development agility [s10, s13, s17], **f23**: Flexibility [s10], **f24**: React to unforeseen changes in the data [s11, s19], **f25**: Memory requirements [s11], **f26**: Handling massive amounts of data [s11, s12, s15], **f27**: Speed performance [s2, s3, s4, s5, s6, s9, s10, s11, s15], **f28**: Variety of ML algorithm choices [s11, s12, s19], **f29**: Explainability [s17], **f30**: Development velocity [s5, s17, s18, s19]

Note on Empty Forces (∅): Please note that some ADDs do not report decision drivers as they are sub-decisions of the respective prior ADD, and share the same forces (see explanations in the remainder of the chapter).

2.5.1 Data Processing

Data processing transforms ingested raw data in an ML workflow into usable data that ML algorithms can process further. Data is central to ML, and the main benefits lie in the processing of large volumes of it at scale [109]. Typical phases of the data processing task involve data selection, transformation and output, as well as the generation, storage and versioning of data artefacts such as features and data subset samples [39].

As with many other ML tasks, data processing can be performed manually or automatically, with the latter offering many potential benefits. The data processing stage also presents various data collection options. Once again, this task may be automated. Data may be ingested manually, in real-time (e.g. streamed), batched, online or offline, with various implications for latency, throughput, reliability and so on – the different options are discussed below. Data labelling (e.g. for classification tasks) is another activity that may be performed manually or automatically, with respective implications. Generated artefacts, such as features, may be stored in various ways, and each option has different implications regarding their respective benefits. Finally, the data processing stage must be triggered. Again, there are numerous options to choose from, including triggering on-demand, on commit, on schedule, on new data availability, on model performance degradation or on changes to the data distribution.

Data Processing Automation Decision A core ADD is *how to automatically process the data used for model building.* A simple option is to provide **no data processing automation**, but the most common automated architecture proposed by practitioners in our sources is a **data pipeline**, i.e. to perform data processing in a dedicated, flexible pipeline. A similar option is an **ETL pipeline**, a type of data pipeline that follows a specific sequence of steps: extract, transform and load. Whilst pipeline architectures

are often mentioned in grey literature, proposals for **data processing components** that do not follow a pipeline architecture also exist.

The primary benefits of these options are supporting *automated data collection* and a potentially high degree of *process and work automation*. These, in turn, may lead to faster data ingestion, shorter model training turnaround times, more frequent deployments, reduced human error and improved *iterative development* and *reproducibility*, not to mention the resulting cost savings. Our sources indicated that the **data pipeline** is the most *flexible* of the automated options, but requires some *tool development effort*, which may represent overhead – especially in the initial phases of a project.

Data Processing Tasks Decision If a **data processing pipeline** or **data processing component** is chosen, the tasks to be performed during automated data processing can be decided in a subsequent decision. Typical tasks performed in automated data processing flows are **extraction**, **transformation**, **preparation**, **validation**, **selection** and **feature engineering**.

Data processing hyperparameter tuning, the tuning of hyperparameters during automated data processing, is a less frequently used method. For example, if an automated featuriser is used during feature engineering, it has hyperparameters that can be tuned. Automation is again possible, e.g. in AutoML (see the discussion on AutoML below).

Feature Persistence and Access Decision The *data processing tasks* decision and its **feature engineering** option generate the features in ML. It is often essential to provide persistence and access to these features – frequently beyond the current run of data processing and model building. One option to achieve this is to **store features in a data store without specific support** for them. The alternative is a dedicated **feature store**, where features are accessible for training in accordance with established standards for definition, storage and access. To facilitate access to features, the feature store can include either a **batch feature API**, a **real-time feature API** or both. Some practitioners also suggest using event-driven abstractions here, e.g. using **event sourcing** as a technique to realise the feature store.

A dedicated **feature store** facilitates easier *feature reuse*, *feature discovery* and *maintainability*. It also helps avoid *training-serving skew*, making sure that the features used for training are the same as the ones used during serving. If dedicated APIs are supported, it can additionally offer better support for *offline* or *online* training and serving. Finally, those APIs can be optimised for *data throughput* (e.g. when large volumes of data or a large number of clients are expected), *low latency* (e.g. in safety-critical systems) or *near real-time support* (e.g. when user interaction is intended).

Data processing in batches or in real-time decision Automated data processing can occur in batches or in real time. Thus, **batch-based data processing** and **real-time, stream-based data processing** are two options for this follow-on decision, to be applied to all automated data processing options.

Real-time, stream-based data processing is especially beneficial for *online* data processing and, unlike *offline* data processing, is beneficial for *low latency* and *near real-time* processing, e.g. when the practitioner wishes to gain fast insights or when a system needs to be able to react instantly to events at runtime. Whilst modern online architectures offer high *data throughput* and *reliability*, the additional runtime processing steps make them inferior to offline architectures in these respects. *Offline* data processing is well supported by **batch-based data processing**, for instance, when processing high volumes of data, performing deep data analyses or when processing speed does not take precedence.

Data ingestion decision Another central ADD is how to ingest data into ML projects or applications to facilitate data processing and, consequently, model building. An initial option is to **ingest data manually**. Alternatively, automated options exist, such as **data ingestion in batches** and **streamed data ingestion**. One can also actively acquire raw data from sources via **data ingestion on request**. It is customary to store ingested raw data, typically in a **raw data store**. The automated options can be abstracted and orchestrated in a dedicated **data ingestor** component, with the two variants **data ingestor queue** (a data ingestor that queues up ingested data, e.g. for online, incremental learning) and **ground truth collector** (a data ingestor collecting data on predictions in production, e.g. with input from users when predictions are wrong). The data ingestor can be used for data labelling. For example, a ground truth collector can support an **automatic data labeller** which is used instead of **manual data labelling**.

Data ingestion for experimental or *offline* learning is often performed using **data ingestion in batches** or by **manual data ingestion**. **Streamed data ingestion** and **data ingestion on request** are suitable for *online* learning or for ingesting data into an ML-based application. **Streamed data ingestion** is ideal for continuously incoming data and offers relatively high *data throughput*, *low latency* and *near-real-time support*. **Data ingestion on request** can receive specific data as needed, but this increases latency and is usually not well suited to achieving high *data throughput*, as each new item of data requires an additional request. **Data ingestion in batches** offers very high data throughput but is *offline* and therefore not well suited for model training or applications that require *low latency* or *near real-time support*. **Manual data ingestion** has even more negative impacts on those forces. Its main advantage is that it does not require *tool development effort* for engineering a custom automated ingestion solution.

Trigger pipelines or ML orchestrator decision ML components, such as data processing pipelines, ML orchestrators and model-building components, can be triggered at various points. For each of these, we are required to decide on how they should be triggered. The first option is an **on-demand trigger** either by an ad-hoc, manual execution of a pipeline or component (e.g. by a human operator) or via a call coming from another pipeline or component.

Another option is an **on commit trigger**, which can start a pipeline when changes are made in a version control system. Some pipelines or components need to run at regular intervals, e.g. daily, weekly or monthly, in which case an **on-schedule trigger** is used.

Triggering can also occur **on the availability of new training data**. Further, pipelines can be triggered by **model performance degradation** or **changes in the data distribution**. Each of these options requires monitoring of the model or data, respectively.

2.5.2 Model Building

Building an ML model involves applying a learning algorithm to the features of a pre-prepared dataset to yield predictions of a specific type, such as qualitative (classification) or quantitative (regression) values. Hyperparameter tuning is the practice of adjusting values used by ML algorithms to optimise their performance metrics. This task may be automated or manual. The actual work involved in model building can be performed using tools such as an integrated development environment (IDE) or a computational notebook or automated using AutoML or a model-building pipeline. Each approach has its advantages and disadvantages, which are discussed below.

Model-Building Decision A core ADD is *how to perform model building in an ML project*. Many practitioner sources claim that providing some level of automation is key to successful production delivery and that many ML projects never progress beyond the experimental phase. Very often, the **model building is performed using tools**, such as computational notebooks or IDEs. The main benefit of **model building using a tool** is that it better supports *interactive prototyping* and *iterative development* than the other options. This option also offers low *tool development effort*, since the tools do not have to be engineered from scratch.

Whilst this is useful for development and experimentation, a **model-building pipeline** is recommended for projects intended for production deployment due to its greater degree of automation. Pipeline abstractions are common for this automation task, but some practitioners also suggest other kinds of **model-building components**, such as a pipeline-architected component or a dedicated **ML orchestrator** that coordinates disparate components at a higher level of abstraction.

Automated model building enables *interchangeability of ML algorithms*, *low latency* in the model-building process, *symmetry between experimental and operational tasks* (in the sense that the same software components are used for both pre-production and production), *reproducibility* of each step in the model-building phase and enhanced support for *production-ready development*. The main drawback is the increased *tool development effort*.

Our sources indicate that, when comparing the two automation options above, the pipeline abstraction is usually preferred, as it is a highly *flexible* architecture that supports greater *development agility*. Pipelines are also known to be *scalable* and *loosely coupled*.

Model-Building Tasks Decision If a **model-building pipeline** or **model-building component** is chosen, the tasks to be performed during automated model building can be decided in a subsequent decision. Common tasks typically performed in such components are **model training**, **data splitting** (e.g. into training and test data), **model validation** and automated or semi-automated **model selection** if multiple models

were built. **Model selection** is usually needed if the practice of **training multiple model versions with different parameters and/or algorithms** is applied. **Model hyperparameter tuning** describes the challenge of selecting optimal hyperparameters for the learning algorithm(s). Some approaches automate **model hyperparameter tuning** (e.g. in AutoML) as a model-building subtask. Our sources describe the tasks of **model packaging** (e.g. in storage or exchange formats). To enable model training with error tolerance, **data checkpoints** can be defined to allow for retraining if a previous attempt failed due to a transient issue (e.g. a timeout). Some automated pipelines or components not only use development tools in early, experimental stages but also integrate them into the automated flows. This integration can be achieved either by using a **development tool export** or, more elegantly, by utilising a **development tool facade** component integrated into the automated flow. This way, the best of both worlds can be achieved.

When and how to train the model decision The *model training* task entails a follow-on decision on *when and how to train the model*. Model training can occur via **batch-based learning** or **incremental learning**. Alternatively, **hybrid batch-based and incremental learning** is possible, where some factors of the model are trained in batches and others incrementally.

Significant benefits of **batch-based learning** lie in *offline* training, especially when *massive amounts of data* are to be handled. Thus, it usually has high *memory requirements* and relatively poor *speed performance*, leading to pre-computation, which is beneficial for *scalability* (provided incremental updates are not needed).

Incremental learning has the opposite effect on those forces. Thus, it is more applicable when a *reaction to unforeseen changes* or the *near real-time support* for model training is required. A downside is that the *variety of ML algorithm choices* supporting **incremental learning** is limited.

2.5.3 AutoML

AutoML is a relatively recent and trending practice in which many tasks typically performed manually by ML engineers, such as data preparation, feature engineering, model training, hyperparameter tuning, model validation and model selection, are automated. To achieve this, AutoML provides search algorithms to identify optimal solutions for tasks within the ML pipeline.

AutoML decision The AutoML decision consists of two parts: (1) *whether AutoML should be applied*; (2) if so, to further decide *which practices should be automated*. In particular, AutoML can automatically preprocess the data during **data preparation**. It can be used during **feature engineering** to generate new features and automatically select meaningful ones. Throughout **model training**, it can be applied to automatically train models, possibly using different parameters and/or ML algorithms by applying automated **hyperparameter tuning** of data processing and model-building components

and then make an automated **model selection**, e.g. based on ML or domain-based metrics. In this context, it can use **model validation** to score the models.

The goals of applying AutoML are, for instance, requiring less manual *tool development effort* and thus supporting higher *development velocity* and *agility*. These goals are achieved through increased *process and work automation*, which also offers further key benefits: via automation, AutoML provides greater *explainability* and *reproducibility* in data processing and model-building tasks, as every step follows precisely specified algorithms. Unfortunately, existing AutoML solutions do not consistently deliver results to the desired quality standards, which often negatively impacts *production-ready development* and additional *tool development effort* for engineering the AutoML solution would potentially also be required in this case.

2.6 Discussion

2.6.1 Research Questions

RQ$_{1.1}$: *Which ADDs are available to choose from in the context of the ML workflow, and what are their corresponding decision options?* As a result of a systematic review of 29 source documents, 10 ADDs, 43 decision options and 30 decision drivers were found to exist in the ML workflow. The decision options can be divided into two major groups: automation-specific decisions and decisions to enhance the benefits of automation. Examples of automation-specific decisions include how to approach *data processing automation*, *model building* and whether to apply *AutoML*. Once an automation level has been specified, the practitioner can then reflect on other choices that depend on the impact of the forces and the specific need. For example, after selecting a **data processing pipeline**, the practitioner needs to decide whether to **process data in batches or real-time**, which *data processing tasks to perform* and how to *persist features*. Similarly, once a **model-building pipeline** is defined, the practitioner can consider *model-building tasks* to automate and the *model training approach*. The choice to automate, therefore, opens new possibilities – some of which may not have been otherwise considered had automation been ignored – that can greatly improve ML processes.

RQ$_{1.2}$: *What are the relations between these decisions and their decision options?* There was an intriguing observation in relation to several decisions resulting from our analysis. The *data processing automation* decision under consideration is a precondition to a series of further decisions, such as which *data processing tasks* to perform, how to *persist and provide access to features*, whether to *process data in batches or real-time* and how to *ingest data*. This dependency means that the decisions concerning information-processing automation should be considered before others when designing the ML workflow. Similarly, the *model-building automation* decision is an antecedent to the decision points regarding *model-building tasks* and the *model training approach* decision. The *AutoML* decision holds a special position, as it is a meta-decision that may affect the solution of tasks at both data processing and model building stages; it is often mentioned alongside various other decisions. This observation implies that, as with data processing automation,

the *AutoML* decision is worth considering earlier on in the planning process. The *how to trigger* decision is a cross-cutting concern that affects many workflow components, rather than being the descendant of a parent decision. Once such recurring decisions are identified, it is rational to prioritise them, as they can also provide solutions to a variety of contexts; thus, they can be regarded as important spheres for further investigation.

RQ$_{1.3}$: *Which decision drivers (forces) are relevant to the decision options?* We sought to understand the decision drivers – or forces – that practitioners consider salient. As a decision option is evaluated, these forces can be categorised into qualitative concerns, including *maintainability*, *modifiability* and *flexibility* and functional concerns, including *data throughput*, *latency* and *algorithmic diversity*. The most common forces that can be met when making the decision choices relate to *automation*, *performance characteristics* and *production readiness*. When it comes to decisions about automation, factors such as *work automation*, *reproducibility* and *production-ready development* tend to emerge. When selecting operational parameters, i.e. **batch-based data processing**, **real-time, stream-based processing** or a form of **data ingestion**, performance-related forces prevail. To select decision options, practitioners must therefore consider opposing forces. As an example, **real-time, stream-based data processing** has strong support for *low latency* and *near real-time support*, but weakly supports *maximum data throughput* compared to **batch-based data processing**. On the other hand, **batch-based data processing** supports *reliability* and *offline processing* but provides poor *real-time support*. These tensions highlight the contextual character of architectural choices: the suitability of a given choice is determined by which forces are most significant in a given project.

2.6.2 Summary

Modelling ADDs as performed in this study yields numerous benefits. In particular, the ADD model can help scientists understand the needs and challenges practitioners face and guide architectural decision-making based on existing practices. The study and its resulting model also open up new avenues for exploration and further research in this domain. One topic that is mentioned in several sources (s4, s6, s13, s21 and s26) is security, which appears in the general context of operations and infrastructure (e.g. serverless, enterprise infrastructure and fully-managed platforms) rather than the ML workflow and thus falls outside the scope of this study.

The validity of the study is premised on the correct application of Grounded Theory and consistent, accurate coding practices. We have endeavoured throughout to apply the method correctly (as documented in our replication package [103]), but mistakes cannot be entirely excluded. Validity may also be threatened by unreliable or omitted sources. Since we continued coding sources until theoretical saturation was reached, this increased the likelihood that a fair representation of the literature was achieved and reduced the risk that relevant sources were not considered.

Bias poses a potentially significant threat to any study, and we have attempted to reduce the risks associated with it. One of the main advantages of applying Grounded Theory to grey literature rather than to interviews conducted by the authors is the prevention of the authors' influence on the results. Furthermore, the authors of this study

were not affiliated with the practitioner sources used, thus further reducing the potential for bias. Finally, the modelling conducted by each author was independently checked by the other author, thus reducing the risk of individual bias during the modelling process.

Not all possible sources were considered in this study, so it cannot be ruled out that some relevant sources were not identified for inclusion, leading to essential factors being overlooked and excluded. This risk cannot be avoided, since it is infeasible to include all available sources. However, since we coded to theoretical saturation, we argue that this limitation does not invalidate our results. Also, please note that Grounded Theory does not claim to achieve completeness [76], [96] but instead describes phenomena that exist. In the same vein, our study does not claim to document "best practices", as these are extensively covered in Serban et al. [106]. The modelled practices, decisions and so on are merely phenomena that have been observed to exist – no further claims are made in this regard, and the options, decisions, and the like may not apply universally in every practical case.

2.7 Conclusion

Our study modelled practitioner methods, ADDs and decision drivers in the field of software engineering and software architecture for ML. The resulting ADD model can help researchers better understand practitioners' needs and challenges, and guide their decisions based on existing practices. The study also opens new avenues for further research in the field, and the design guidance provided by our ADD model can also help reduce design effort and risk. Future work could involve using our findings to provide automated design guidance to ML engineers.

3 Architectural Design Decisions for Machine Learning Deployment

Deploying ML models to production is challenging, partly because of misalignment between the software engineering and ML disciplines and partly because of potential knowledge gaps amongst practitioners. To reduce this gap and inform decision-making, we conducted a qualitative investigation into the technical challenges faced by practitioners, drawing on a review of grey literature and applying the Straussian Grounded Theory research method. We modelled current practices in ML, resulting in an ADD model based on current practitioner understanding of the domain and a subset of the decision space. We identified seven ADDs, various relations between them, 26 decision options and 44 decision drivers across 35 sources. Our results aim to help bridge the gap between science and practice, increase understanding of how practitioners deploy their solutions and support practitioners in their decision-making.

3.1 Introduction

Software engineering, software architecture, and ML are all established disciplines, but a significant disparity has emerged between software engineering and software architecture on the one hand and ML on the other, even though ML often involves software engineering activities such as development, maintenance, and deployment [38], [39], [40]. ML practitioners have various ADD options to choose from. Since software engineering and software architecture approaches to ML are still at a relatively early stage compared to traditional software engineering and software architecture practices, in addition to the lack of systematic approaches to solution building and systems design, it may be hard for practitioners to know which options are available to choose from in a given situation.

We conducted a qualitative grey literature study [95], [97], based on Grounded Theory [65], [76], [96], [110], to formalise current practitioner understanding and architectural concepts of ML solution deployment. Other detailed aspects of ML engineering, such as data processing, big data and development environments, were not explicitly part of this study. We cover data processing as part of the ML workflow similarly in prior work [23] in this doctoral thesis, and may address the remaining topics in future work.

This chapter corresponds to $\mathbf{P_2}$, presents $\mathbf{RC_2}$ and sets out to address the following research questions as part of $\mathbf{RQ_1}$ and $\mathbf{RP_1}$:

$\mathbf{RQ_{1.4}}$ *Which ADDs are available to choose from in the context of deployment for ML, and what are their corresponding decision options?*

RQ1.5 *What are the relations between these decisions and their decision options?*

RQ1.6 *Which decision drivers (forces) are relevant to the decision options?*

The result was a formal model of ADDs, decision options, decision drivers and their relationships in the domain of ML deployment, implemented in Python, that serves to guide practitioners and advance the scientific knowledge of their practices, concerns and understanding in the field of ML deployment.

We discuss related studies in the field in Section 3.2 and describe our research method in Section 3.3. After that, in Section 3.4, we present the various ADDs, decision options, decision drivers and their relationships, which represent the results of our empirical study. In Section 3.5, we interpret our results and consider possible threats to validity before finally concluding in Section 3.6.

3.2 Related Work

There appear to be relatively few standardised methods and processes in software engineering for ML. Thus, ML engineers who wish to apply professional software engineering techniques must contend with numerous engineering and architectural decisions.

Washizaki et al. [104] conducted a systematic literature review that categorised and detailed a collection of software patterns and anti-patterns for ML. Like our study, theirs is based on grey literature, but it also includes scientific literature. In contrast to ours, their work focuses on ML pipeline and software development patterns rather than deployment patterns, which is the focus of our work.

Sculley et al. [38] describe the technical debt incurred when maintaining production ML systems, including entanglement, data dependencies, configuration challenges, reproducibility debt and system anti-patterns. Some of these factors are essential motivations for our work, but in contrast to our work, Sculley et al. do not provide detailed ADDs for ML deployment.

Lwakatare et al. [39] conducted an empirical investigation into the software engineering challenges of ML systems. They devise a taxonomy of common issues, including data dependency management, deployment difficulties and challenges in result reproduction, without formally modelling design decisions.

Bosch et al. [105] provide an overview of the software engineering challenges associated with ML solutions, including model management, deployment, data pipeline challenges, monitoring, logging, design methods and data quality management. They also identify unresolved research areas, including the creation of distributed models, distributed data storage, data generation and automated experimentation. Whereas our work concentrates on concrete ADDs for deployment, theirs focuses on identifying challenges (which partially appear in our decision drivers) and categorising relevant practices at a higher level of abstraction than we do.

Nascimento et al. [40] provide a comprehensive systematic literature review on software engineering for ML, document the state of the art and identify open challenges, mainly in testing, AI software quality and data management. They found that most of the proposed

software engineering practices are guidelines, lessons learned and tools. We propose a formalised guidance model in the form of ADDs for the deployment of ML models, a field not yet adequately covered in the literature.

Mäkinen et al. [111] studied the importance of MLOps to practitioners for rapid deployment in ML via a survey. They discovered that most respondents were focused on utilising data effectively, the initial building and deployment of models and that MLOps is only beneficial for retraining and redeploying models. Unlike our approach, they did not develop an ADD model for ML deployment. Despite their findings and the overlap with the subject of this study, our research indicated that MLOps is a complex topic and includes many non-deployment-related ADDs.

Our approach was to reduce the gap between science and practice by studying methods and techniques documented by ML practitioners in deployment contexts. We formally modelled ADDs, decision options, practices, decision drivers and their relations. Valuable insights were gained, which may guide practitioners in selecting suitable solutions to recurring design decisions, help practitioners overcome challenges, mitigate problems and avoid unsuitable decisions. In this study, we restrict ourselves to decisions associated with the *deployment* of ML models. To our knowledge, this is one of the first studies of its kind in the field.

3.3 Research Method

We conducted a grey literature study as a systematic investigation into practitioners' understanding of ML model deployment, using practitioner sources exclusively. Applying Straussian Grounded Theory [76], [96], we coded the phenomena encountered in our sources and developed a formal theory, coded as a model in Python with UML-based visualisations. We previously successfully applied Grounded Theory, combined with a grey literature study, in other studies [23], [98], [99], and briefly describe the method below.

Initially, we used search engines (e.g. Google, StartPage and DuckDuckGo) and topic portals (e.g. InfoQ and DZone) to search for practitioner sources consisting of blog posts, presentations and videos. Our initial source search term was "ML deployment". Sources were considered relevant if they were not marketing a business or product. The authors checked each other's selection for suitability.

We open-coded each source in depth and performed axial and selective coding, formally modelling each ADD, ADD relation and decision driver in Python-based models and generating automated UML models in PlantUML [79]. Further sources were actively sought during the coding process based on the preceding findings. This search involved continuous consideration of the topics needing to be coded and their potential contribution to the model.

We continued coding sources until theoretical saturation (i.e. "*the point in category development at which no new properties, dimensions, or relationships emerge during analysis*" [96]) was reached after 35 sources. See Table 3.1 for the full list of sources. The result was a formal UML-based ADD model that covered decision options, their relationships and relevant decision drivers.

Table 3.1: List of Knowledge Sources Included in the Study

ID	Title	Source Type	Example	Source Code
s1	How to power up your product by machine learning with python microservice, pt. 1	Practitioner Audience Article	True	False
s2	Architecting a Machine Learning Pipeline: How to build scalable Machine Learning systems – Part 2/2	Practitioner Audience Article	True	False
s3	Some Thoughts on Modularization in Machine Learning	Practitioner Audience Article	False	False
s4	Productionizing Machine Learning with a Microservices Architecture	Presentation Video	False	False
s5	Microservices Suck for Machine Learning (and what we did about it)	Practitioner Audience Article	True	True
s6	MLOps: Continuous delivery and automation pipelines in machine learning	Practitioner Audience Article	True	False
s7	Composing Deep-Learning Microservices for the Hybrid Internet of Things	Practitioner Audience Article	True	False
s8	Continuous Intelligence: Moving Machine Learning Application into Production Reliably	Slides	True	False
s9	Architecture of a real-world Machine Learning system	Practitioner Audience Article	True	False
s10	Architecting a Machine Learning System for Risk	Practitioner Audience Article	True	False
s11	Architecting a Scalable Real Time Learning System	Practitioner Audience Article	True	False
s12	System Architectures for Personalization and Recommendation	Practitioner Audience Article	True	False
s13	Architectural thinking in the Wild West of data science	Practitioner Audience Article	True	False
s14	Machine Learning Architecture: The Core Components	Practitioner Audience Article	False	False
s15	Scalable Software and Big Data Architecture – Big Data and Analytics Architectural Patterns	Practitioner Audience Article	False	False
s16	Machine Learning in Production: Software Architecture	Practitioner Audience Article	True	False

Continued on next page...

ID	Title	Source Type	Example	Source Code
s17	AutoML	Practitioner Audience Article	False	False
s18	AutoML is Overhyped	Practitioner Audience Article	True	False
s19	Three Levels of ML Software	Practitioner Audience Article	True	False
s20	MLOps: Methods and Tools of DevOps for Machine Learning	Practitioner Audience Article	False	False
s21	MLOps: What It Is, Why it Matters, and How To Implement It (from a Data Scientist Perspective)	Practitioner Audience Article	False	False
s22	MLOps Principles	Practitioner Audience Article	False	False
s23	Machine Learning Monitoring: What It Is, and What We Are Missing	Practitioner Audience Article	False	False
s24	Automated monitoring of your machine learning models with Amazon SageMaker Model Monitor and sending predictions to human review workflows using Amazon A2I	Blog Post	False	False
s25	MLOps: Model management, deployment, and monitoring with Azure Machine Learning	Practitioner Audience Article	False	False
s26	The Pros and Cons of Using Jupyter Notebooks as Your Editor for Data Science Work TL;DR: PyCharm's probably better	Practitioner Audience Article	False	False
s27	10 reasons why data scientists love Jupyter notebooks	Practitioner Audience Article	False	False
s28	5 reasons why jupyter notebooks suck	Practitioner Audience Article	False	False
s29	Jupyter Notebook is the Cancer of ML Engineering	Practitioner Audience Article	False	False
s30	Comparing Data Version Control Tools – 2020	Blog Post	False	False
s31	How to test your ML models from hypothesis to production	Blog Post	False	False
s32	Evaluate ML Classifier Performance using Statistical Hypothesis Testing in Python	Blog Post	True	False
s33	Testing Your Machine Learning Pipelines	Blog Post	True	False

Continued on next page...

ID	Title	Source Type	Example	Source Code
s34	Performance Testing ML Serving APIs With Locust	Blog Post	True	False
s35	Machine Learning at Scale with Parallel Processing	Blog Post	True	False

3.4 ADDs

This section presents our study results as ADDs derived from practitioner views and practices sourced from grey literature within the context of deploying ML models. Table 3.2 shows an overview of the various ADDs, the number of sources, the decision options, the practitioner sources[1][2] and the decision drivers (forces) with their impacts[3]. These are described in this section.

Table 3.2: Study Results: Overview of ADDs, Number of Sources, Decision Options, Practitioner Sources and Related Decision Drivers

ADDs	#	Decision Options	Practitioner Sources	Decision Drivers
How to approach deployment of ML models?	14	1. No automated deployment	s1, s2, s3, s4, s6, s15, s16, s26, s27, s28, s35	f1(--), f2(-), f3(--)
		2. Build, test, and deploy models to be served and other system components in a semi-automated fashion by deploying pre-prepared pipelines	s1, s2, s4, s6, s19, s20	f1(+), f2(+), f3(++)
		3. Build, test, and deploy ML models based on CI/CD pipeline automation	s1, s2, s6, s19, s20, s25	f1(++), f2(++), f3(++)
How to automate integration and delivery in an ML context?		1. No integration or delivery automation	s1, s6, s20	f1(--), f2(-), f4(-), f5(--), f6(--), f7(--), f8(-), f9(--), f10(--), f11(-), f12(--), f13(--)
		2. Build and deployment scripts	s1, s9, s10	f1(+), f2(+), f4(+), f5(++), f6(-), f7(-), f8(o), f9(-), f10(+), f11(o), f12(-), f13(-)

Continued on next page...

[1] The following colour scheme in Table 3.2 indicates the source frequency: < 5%, < 10%, < 20%, < 35%, < 50%, < 70%, ≥ 70%.

[2] Source archive URLs may be found in our replication package [112].

[3] Force impacts in Table 3.2 range from "very negative" (--) to "very positive" (++) based on our interpretation of the grey literature sources.

Design Decision	#	Decision Option	Practitioner Sources	Decision Drivers
	16	3. CI/CD pipeline	s1, s2, s4, s6, s8, s9, s10, s13, s20, s25, s31, s32, s33	f1(++), f2(++), f4(++), f5(++), f6(++), f7(+), f8(+), f9(++), f10(++), f11(++), f12(++), f13(++)
		4. ML orchestrator	s4, s6, s8, s9, s14, s21, s29	f1(+), f14(+)
Which tasks can be performed by a delivery pipeline or component?	19	1. Packaging	s2, s4, s6, s25	∅
		2. Testing	s2, s4, s5, s6, s31, s33, s34	∅
		3. Building	s4, s6, s8	∅
		4. Deployment	s4, s6, s8, s13, s17, s18, s19, s20, s23, s24, s25	∅
		5. Containerisation	s4, s7, s8, s9, s11, s19	∅
How to trigger an ML pipeline or orchestrator?	14	1. On-demand trigger	s2, s4, s6, s9, s12, s21, s22	∅
		2. On commit trigger	s4, s8, s22, s25	∅
		3. On schedule trigger	s1, s2, s6, s9, s12, s14, s21, s22, s33	∅
		4. On availability of new training data trigger	s6, s21, s22	∅
		5. On model performance degradation trigger	s6, s19, s21, s23	∅
		6. On changes to the data distribution trigger	s6, s21, s22	∅
How to version data?	8	1. No data versioning	s4, s21, s22, s30, s31	f10(--), f15(-)
		2. Data version repository	s4, s8, s20, s22, s25, s30, s31	f4(++), f10(++), f15(++)
		3. Data versioning in code repository	s4, s22, s30, s31	f4(--), f10(--), f15(--), f16(--)
Which model versions should be deployed and how?	7	1. Single model in production	s2, s6, s13	f17(--), f18(--)
		2. N versions in production	s2, s4, s6, s9, s20, s25	f17(++), f18(++), f19(++)
		3. Rollback to previous model version	s4, s6	f17(+), f18(+)

Continued on next page...

Design Decision	#	Decision Option	Practitioner Sources	Decision Drivers
Should MLOps be applied and, if so, when?	8	1. No MLOps	s4, s6, s19, s20, s21, s22, s25	f1(o), f2(o), f4(o), f5(o), f6(o), f10(o), f12(o), f13(o), f14(o), f19(o), f20(o), f21(o), f22(o), f23(o), f24(o), f25(o), f26(o), f27(o), f28(o), f29(o), f30(o), f31(o), f32(o), f33(o), f34(o), f35(o), f36(o), f37(o), f38(o), f39(o), f40(o), f41(o), f42(o), f43(o), f44(o)
		2. MLOps	s4, s6, s19, s20, s21, s22, s23, s25	f1(+), f2(+), f4(+), f5(+), f6(+), f10(+), f12(+), f13(+), f14(+), f19(+), f20(+), f21(+), f22(+), f23(+), f24(+), f25(+), f26(+), f27(+), f28(+), f29(+), f30(+), f31(+), f32(+), f33(+), f34(+), f35(+), f36(+), f37(+), f38(+), f39(+), f40(+), f41(+), f42(+), f43(+), f44(+)

Forces Codes/Sources: **f1**: Process and work automation [s1, s8, s15, s18, s20, s22], **f2**: Iterative development [s1, s4, s22], **f3**: Observability [s4, s8, s12, s19], **f4**: Development velocity [s5, s17, s18, s19, s25, s26, s29, s30, s31], **f5**: Deployment velocity [s25, s29], **f6**: Dependable releases [s20], **f7**: Reliability [s2, s5, s8, s10, s23], **f8**: Independent development [s1, s3, s4, s5, s6, s7, s19], **f9**: Modifiability [s5, s6, s8, s33], **f10**: Reproducibility [s6, s8, s13, s17, s20, s21, s22, s25, s26, s27, s28, s29, s30, s31, s33], **f11**: Testability [s8, s28], **f12**: Auditability [s8, s25, s33], **f13**: Continuous improvement [s8, s21], **f14**: Ability to identify whether data has changed over time [s20, s24, s25], **f15**: Handling large data files [s30, s31], **f16**: Flexibility [s10, s26, s27, s30, s31], **f17**: Safe transition between production models [s2, s6], **f18**: Rolling upgrades [s4, s6], **f19**: Shadow mode testing [s6, s20, s25, s31], **f20**: Division of labour [s1, s3, s4, s5, s7, s20], **f21**: Reduced time to market [s20, s21, s29], **f22**: Better user experience [s20, s21], **f23**: Better customer satisfaction [s20], **f24**: Higher quality of predictions [s20, s21, s25], **f25**: Ability to recognise model degradation [s20, s24], **f26**: Ability to address model degradation [s20, s24], **f27**: Improve model management [s20, s21], **f28**: Change tracking [s20, s26], **f29**: Simplified deployment [s20, s21, s29], **f30**: Align models with business needs [s21, s23, s33, s34], **f31**: Align models with regulatory requirements [s21, s23, s25, s33], **f32**: Explainability [s17, s21, s25, s29, s33], **f33**: Reusability [s7, s13, s19, s21, s25, s26], **f34**: Infrastructure costs [s5, s21], **f35**: Development agility [s10, s13, s17, s21, s26, s27, s28], **f36**: Model consistency [s21, s25, s33], **f37**: Unlock new sources of revenue [s21], **f38**: Save time [s21], **f39**: Reduce cost [s21, s23, s34], **f40**: Management agility [s21], **f41**: Experimental operational symmetry [s6, s13, s21], **f42**: Modularity [s7, s10, s16, s19, s21, s25], **f43**: Rapid experimentation [s21, s25, s26], **f44**: Reduce technical debt [s22, s26, s29]

3.4.1 Deployment Approach Decision

Deployment in an ML context is the process of making a trained ML model available so that clients can use its predictions. When planning to deploy ML models, a decision

must be made about the level of automation to apply. Our sources identified three initial decision options addressing automated deployments, each encompassing varying levels of automation.

As illustrated in Figure 3.1, the trivial decision solution is to opt for **no automated deployment**; however, this option hinders *iterative development* and precludes *process and work automation*, along with reducing *observability*. An alternative deployment decision option that offers at least some degree of automated delivery is **building, testing and deploying models to be served, along with other system components in a semi-automated fashion by deploying pre-prepared pipelines**. This process is considered semi-automated because although the pipeline execution can be automated, the pipeline must be manually created whenever it changes in any aspect. The advantage of adopting this approach is increased support for *process and work automation* and *iterative development*, as well as enhanced *observability*.

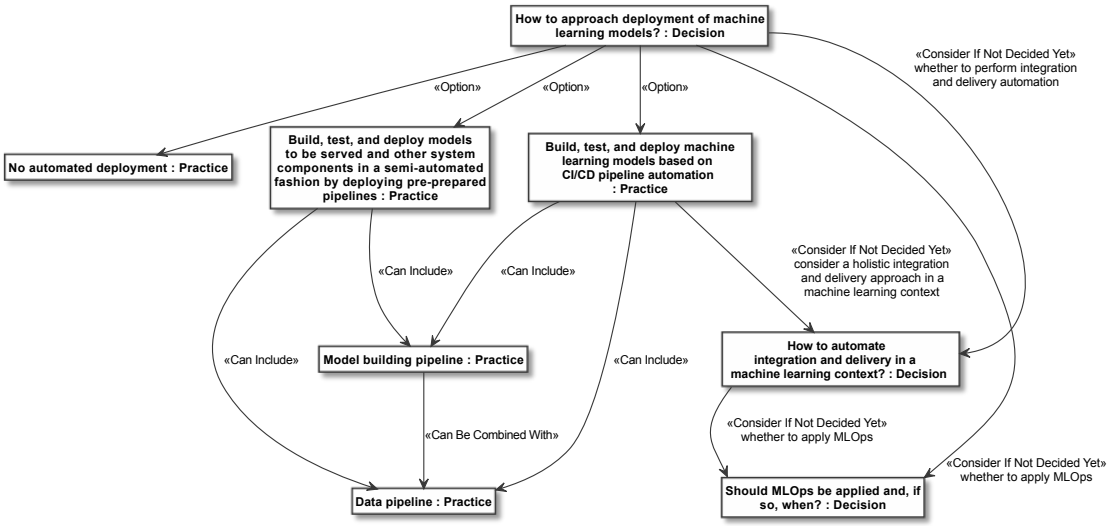

Figure 3.1: Deployment approach decision.

An option that yields greater automation is to **build, test and deploy ML models based on CI/CD pipeline automation**. "CI/CD" stands for "continuous integration/continuous delivery". CI is "*the practice of building and running automated tests against every change you make to your application so you can ensure that your software is always in a working state*" [22], whereas CD "*provides the ability to release new, working versions of your software several times a day*" [22]. Thus, CI/CD is understood as the combination of both approaches. Applying this level of automation leads to a dramatic increase in support for all three forces. Noteworthy is that both the semi-automated and CI/CD pipeline decision options can both include a **data pipeline** (automation of the data processing steps involved in building an ML model), a **model building pipeline** (automation of the training and evaluation steps of building an ML model) or both, since these practices may also be combined.

A related decision to consider, either from the outset or when evaluating whether to apply CI/CD pipelines, is a more inclusive approach, specifically *how to automate integration and delivery in an ML context*. We shall consider this decision in Section 3.4.3. Finally, *should MLOps be applied and, if so, when* is another related decision and is discussed in Section 3.4.2.

3.4.2 MLOps Decision

Should MLOps be applied and, if so, when is a recurring decision throughout this study. MLOps has become increasingly relevant to practitioners [111] and encompasses techniques for the rapid deployment of ML models. It includes data-related practices such as **data sourcing**, **data analysis** and **data labelling**; model-related practices like **model benchmarking**, **model training** and **model validation**; and various other practices including **experiment tracking**, **hardware scaling** and **model service profiling**. As evident in Table 3.2, MLOps is a holistic approach that is associated with 35 decision drivers out of the 44 we discovered and thus represents an important design decision.

3.4.3 Automatic Integration and Delivery in an ML Context Decision

When deciding *how to automate integration and delivery in an ML context*, the simplest solution is to opt for **no integration or delivery automation**. Several practitioners strongly advise against this approach due to its detrimental impact on many forces, including *process and work automation*, *iterative development*, *deployment velocity*, *dependable releases*, *reliability*, *independent development*, *modifiability*, *reproducibility*, *testability*, *auditability* and *continuous improvement*.

An alternative option for continuously automating deployment processing is to use a **CI/CD pipeline**, which was previously considered for the initial *deployment approach decision*. Our sources recommend this, particularly regarding the aforementioned forces. **CI/CD pipelines** are related to several other practices. For instance, they can be combined with an **ML orchestrator** to coordinate pipelines and experiments. An ML orchestrator is a component that can take responsibility for automating dataset preparation and cleansing, or for coordinating model development across specific data sources and features.

Applying a **CI/CD pipeline** enables *automated provisioning*, that is, automation of the actual deployment step. The **CI/CD pipeline** can use a **model registry** to access models and model candidates. Pipeline parameters and versions can be stored in an **ML metadata store** where, e.g. data and parameters can be tracked so that the practitioner can understand how specific models were built. It is worth noting that the two aforementioned practices entail a related design decision (*how to store data used in ML applications*), but this decision shall not be explored further since it is concerned with operational aspects of ML applications and no longer falls within the scope of this study.

An alternative solution is to manually perform automated deployments using **build and deployment scripts**, which is beneficial for *process and work automation*, *iterative development*, *deployment velocity*, *dependable releases* and *reproducibility*. The **ML or-**

chestrator (previously described in relation to **CI/CD pipelines** and for which two forces are described in the literature) is yet another potential decision option. Using an orchestrator allows monitoring data changes, which is beneficial for *identifying whether data has changed over time*. Such monitoring can be advantageous, as it enables the data and model-building pipelines to be triggered as early as possible based on *changes to the data distribution* (see Section 3.4.5), thereby reducing the likelihood that obsolete models remain in production longer than necessary. *Process and work automation* is also improved by using an orchestrator: by not running everything manually, we reduce the scope for human error and free up engineers for other tasks.

Two related practices (**build and deployment scripts** and **CI/CD pipeline**) invite the practitioner to consider *which tasks can be performed by a delivery pipeline or component*. This decision shall be described in Section 3.4.4. Various other top-level decisions present themselves for consideration, e.g. *how to trigger an ML pipeline or orchestrator*, *how to version data* and *which model versions should be deployed and how*. These decisions shall likewise be described below. Finally, as shown in Figure 3.2, every decision involving automation also suggests considering **MLOps** as a design decision.

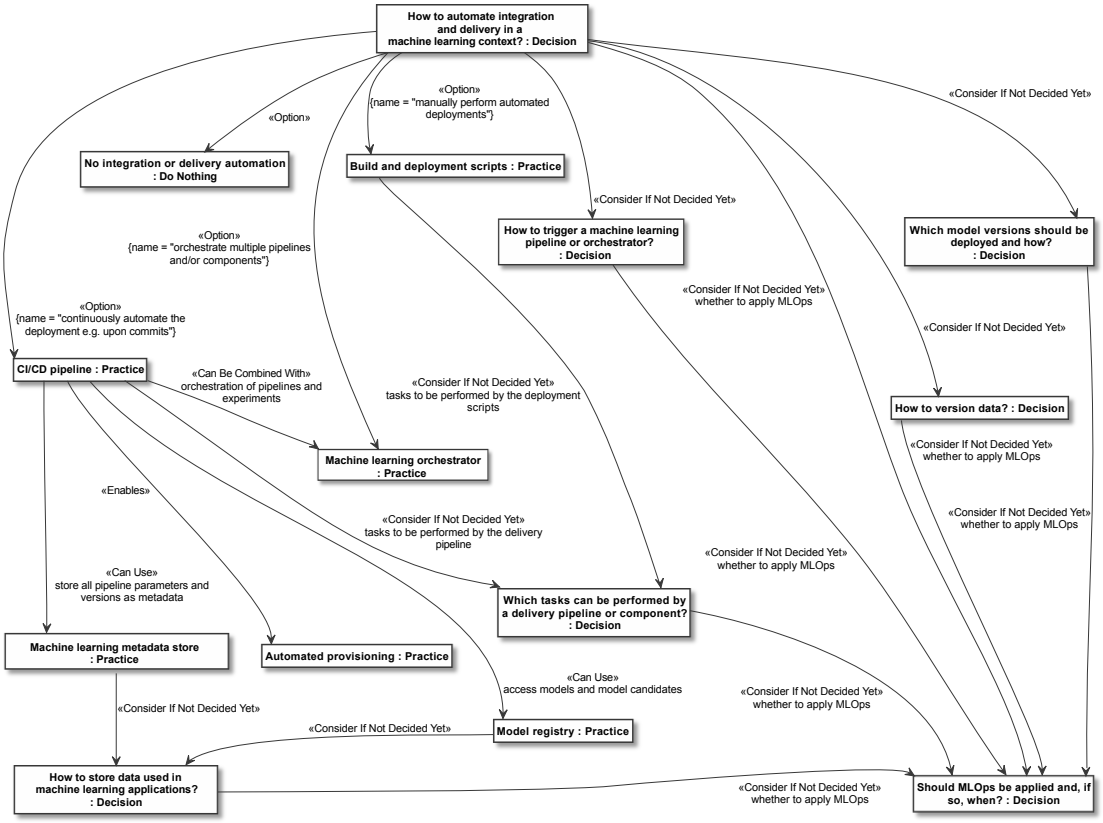

Figure 3.2: Automated integration and delivery in an ML context decision.

3.4.4 Delivery Tasks Decision

When planning how to deploy ML models, the practitioner needs to consider *which tasks can be performed by a delivery pipeline or component*, as presented in Figure 3.3. For this ADD, the list of forces in Table 3.2 for the corresponding decision options described below is empty (∅) because they are inherited from two parent decision options: **CI/CD pipeline** and **build and deployment scripts**.

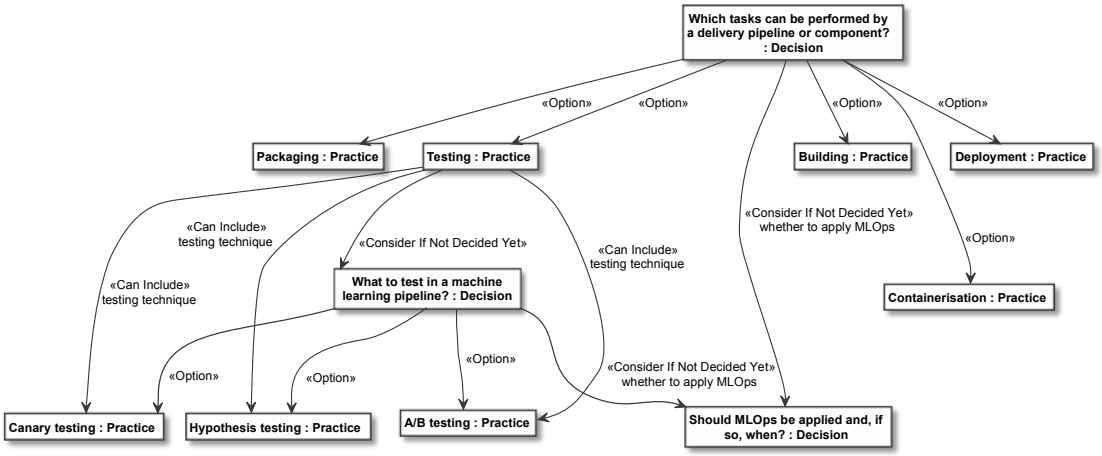

Figure 3.3: Delivery tasks decision.

One essential *task that a delivery pipeline or component can perform* is **building**, which, in the context of delivery tasks, may involve creating models, container images, executables, services or ML pipelines.

Another vital task is **packaging**, which involves assembling build artefacts for further use. **Packaging** goes hand in hand with **containerisation**, which uses, e.g. Docker [33] to prepare application images and execute them in a containerised environment.

Testing is yet another delivery task. It may include various techniques, such as **canary testing** (initially releasing new versions of an application to a subset of traffic/users), **A/B testing** (randomised, statistical comparison of versions) and **hypothesis testing** (a comparison of ML algorithms before deployment). The related decision of *what to test in an ML pipeline* also uses these techniques. This decision will not be explored further, as it is not specifically concerned with deployment. Deciding whether *MLOps should be applied and, if so, when* can be considered from the outset or when deciding *what to test in an ML pipeline*.

3.4.5 Trigger Pipeline or Orchestrator Decision

ML pipelines and orchestrators ("components") can be initialised through triggers. Triggers vary in nature and may, for example, be based on events during the development process, feedback from a live system or be scheduled or manual. The corresponding ADD is *how to trigger an ML pipeline or orchestrator*, presented in Figure 3.4. As in Section 3.4.4,

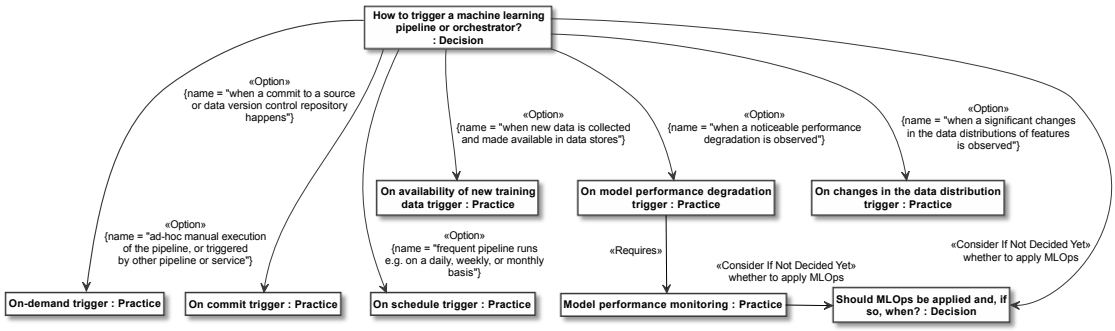

Figure 3.4: Trigger pipeline or orchestrator decision.

the decision options' forces for this ADD are inherited from a parent; in this case, the ADD *how to automate integration and delivery in an ML context*. Thus, the list of forces for the following trigger-specific decision options is empty (∅) in Table 3.2.

- An **on-demand trigger** initiates a pipeline or orchestrator via manual execution, such as by a human operator.
- An **on commit trigger** activates components following a commit event in a source control management or data version repository system.
- An **on schedule trigger** starts a component at a given time or with specified regularity.
- If **new training data** becomes available, this may also prompt a component to start.
- If **model performance degradation** is detected, this event may also serve as a component trigger. Note that **model performance monitoring** is a prerequisite for this option.
- **Changes to the data distribution** may trigger a component when statistical changes to any data set used to train the model cross a predefined threshold.

When evaluating this decision and the **model performance monitoring** practice required for detecting **model performance degradation**, deciding whether *MLOps should be applied and, if so, when* can be considered at this stage, if the practitioner has not already done so.

3.4.6 Data Versioning Decision

Managing data is central to all phases of ML model development. Furthermore, deciding *how to version data* (presented in Figure 3.5) is directly related to model deployment since a significant design decision – discussed in Section 3.4.7 – surrounds *which model versions should be deployed and how*. Three decision options were identified in the literature, and

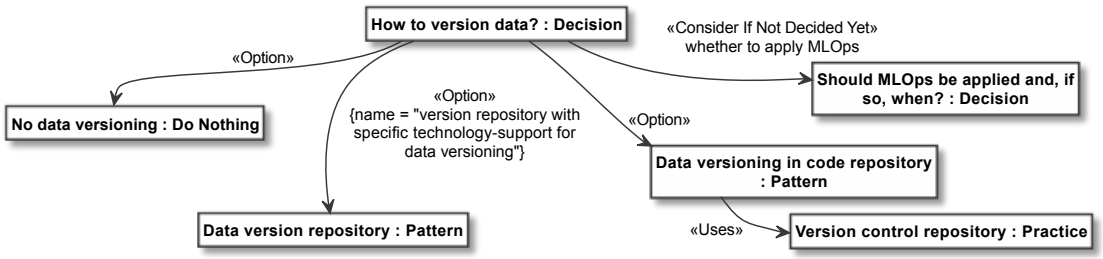

Figure 3.5: Data versioning decision.

it is crucial to carefully consider these practices, as not all forces were described for each practice. For instance, *handling large data files* and *reproducibility* are defined for all three practices, whereas *development velocity* was defined for two practices and *flexibility* was only defined for one.

The simplest option is **no data versioning**, but practitioners consider it undesirable for *handling large data files*, which is not ideal since ML routinely uses them. It is also detrimental to *reproducibility*, as it becomes highly challenging to track data changes when practising either technique.

Data versioning in a code repository is a typical pattern that uses version control repositories such as Git to store ML data files. This approach to version control may initially appear to be appealing and intuitive since such repositories are excellent for versioning source code. However, several drawbacks may arise with such systems when *handling large data files*, which are typically used in ML. Such code versioning systems are not suited to versioning large files and do not aid *reproducibility*, since data file versions are hard to keep track of when they are updated simultaneously, and the *development velocity* is hindered because code versioning systems are slow when *handling large files*. Finally, the tight coupling with the same system used for source control reduces *flexibility*. This practice is considered unsuitable in the literature across all four practices.

Our sources indicate that if the practitioner wishes to increase support for *handling large data files*, *reproducibility* and *development velocity*, using a **data version repository** designed to support large files in an ML context, such as DVC, appears to be an ideal approach. Note that no impact on *flexibility* was documented in our sources when using a **data version repository**. Finally, the only related decision to consider, if not already decided, is whether *MLOps should be applied and, if so, when*.

3.4.7 Deploying Model Versions Decision

As mentioned previously, another significant ADD to consider is *which model versions should be deployed and how*. This ADD is depicted in Figure 3.6. The sources used in this study identified three main decision options, each related to two possible forces. The decision to deploy a **single model in production** is the most straightforward option. Still, it is unsuitable if the practitioner is aiming for **safe model transitions** or **rolling upgrades**, primarily because of the downtime required to swap out a single

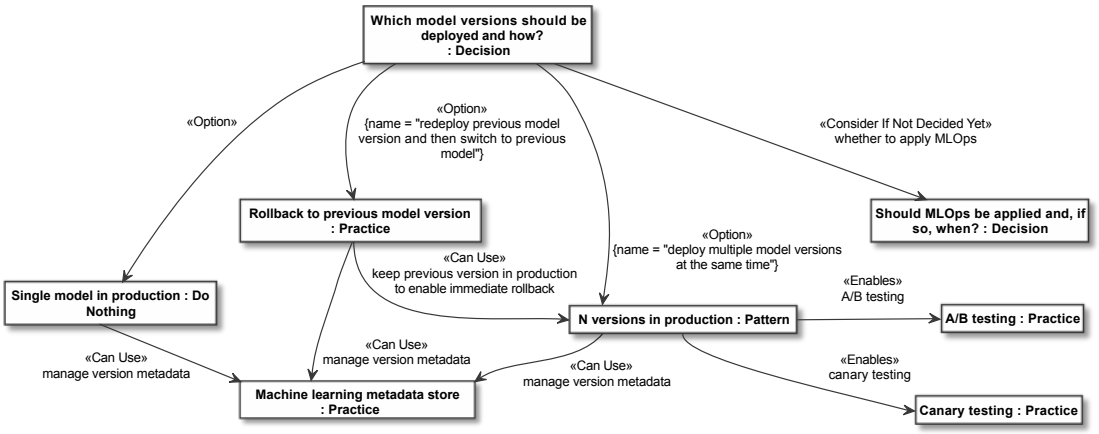

Figure 3.6: Deploying model versions decision.

model version and the lack of availability of multiple model versions. The practice of **rolling back to a previous model version**, on the other hand, is more suited to both *safe model transitions* and *rolling upgrades*, as the practitioner can easily revert to a previously deployed model version. Finally, **n versions in production** is a pattern highly recommended in the literature for both *safe model transitions* and *rolling upgrades*, as it enables smooth swapping out of model versions without downtime. This pattern is also suited to *shadow mode testing*, enabling *A/B testing* and *canary testing*, thereby increasing confidence that new model versions are production-ready. Notably, **n versions in production** does not appear to impact *reliability*.

All three practices use a **ML metadata store**, which was previously encountered when considering *how to automate integration and delivery in an ML context*. The practitioner may also wish to consider *MLOps* if this has not been decided.

3.5 Discussion

3.5.1 Discussion of the Research Questions

RQ$_{1.4}$: *Which ADDs are available to choose from in the context of deployment for ML, and what are their corresponding decision options?* After analysing 35 sources, we discovered evidence for 7 ADDs, various relations between them, 26 decision options and 44 decision drivers in the context of deployment for ML. We note that the decision options can be grouped into two main categories: automation-specific decisions and decisions to enhance the benefits of automation.

Examples of automation-specific decisions include *how to approach deployment of ML models*, *how to automate integration and delivery in an ML context* and *should MLOps be applied and, if so, when*. Once a level of automation has been selected, the practitioner may consider other decisions, e.g. what the automation process should support, based on associated force impacts and specific requirements. For instance, the practitioner may

consider *which tasks* they would like their delivery pipeline or orchestrator to perform and how to *trigger* it. Furthermore, the practitioner may consider *how to version data* and *which model versions should be deployed and how*. Opting for automation opens up new possibilities – many of which might not have even been considered if automation had been disregarded – that can aid ML considerably.

RQ1.5: *What are the relations between these decisions and their decision options?* A curious phenomenon emerged in connection with a particular decision. *MLOps* appears multiple times as a related decision for all six other decisions. *MLOps* is so prevalent throughout our findings that it may be worth considering *MLOps* first when planning the deployment. Giving precedence to such recurring decisions makes sense if identified, as they can lead to solutions across multiple contexts. This discovery also indicates that MLOps may be a significant area for further research.

Similarly, when considering decision options, **CI/CD pipelines** appeared more than once. Interestingly, when considering **CI/CD pipelines** in the context of *how to approach deployment*, the practitioner is invited to consider **how to automate integration and delivery**, which also includes the **CI/CD pipeline** option and is a more comprehensive approach to automation. This logical progression corresponds to the natural thought process of a practitioner as they search for and choose solutions. Upon considering a potential solution, the practitioner is led towards an all-encompassing approach that opens up new possibilities.

RQ1.6: *Which decision drivers (forces) are relevant to the decision options?* Our research has helped us identify the decision drivers (forces) that practitioners are most concerned with. When evaluating a given decision option, these are relevant factors and fall broadly into two categories: **qualitative** (e.g. *modifiability*) and **functional** (e.g. *iterative development*). Forces (codes of which can be cross-referenced in Table 3.2) can also be categorised by the aspects they support as follows:

- **Automation**: f1.
- **Development**: f2, f4, f8, f9, f13, f15, f18, f35, f44.
- **Understanding**: f3, f10, f12, f32.
- **Deployment**: f5, f6, f16, f17, f29.
- **Performance**: f7, f22.
- **Quality**: f11, f20, f28.
- **ML-specific needs**: f14, f23, f24, f25, f26, f27, f33, f36, f41, f42, f43.
- **Business needs**: f19, f21, f30, f31, f34, f37, f38, f39, f40.

The top three categories, based on the number of forces, were support for *ML-specific needs*, *business needs* and *development*. Out of all the decision options, using a **CI/CD pipeline** for the *automated integration and delivery* decision was mentioned most (in 13 sources), whereas **rolling back to a previous model** when deciding *which model*

versions should be deployed and how was mentioned least (in just two sources). For the decision options in Table 3.2 without force relations, the reader is reminded that any out-of-scope forces were excluded from the study, and these options inherit forces from their parent decisions and practices.

3.5.2 Threats to Validity

Stol et al. [113] describe "method slurring", which encompasses common methodological errors that undermine Grounded Theory and may generally occur when one claims "*to use a research method without actually following its guidelines*" – precisely (in the case of Grounded Theory) when researchers "*adopt an arbitrary subset of Grounded Theory practices that are not recognisable as Grounded Theory*". The likelihood of this study exhibiting method slurring is low, as we include all the steps necessary for a Grounded Theory study according to Stol et al. [113] (simultaneous data collection and analysis, constant comparison, coding, memoing and theory development). We did not collect our data before beginning analysis; neither did we collect or categorise data according to an existing theory, nor did we base our analysis on seed categories or preconceived analytical frameworks. We also specify the exact version of Grounded Theory we applied, i.e. Straussian Grounded Theory.

Our study employs a subjective methodology, so it cannot be ruled out that our interpretation of the sources involved a certain degree of bias. This threat cannot be eliminated, but since both authors iteratively cross-checked each other's activities, we assert that any bias or inaccuracies were reduced.

A methodological threat to validity may arise when using unreliable or irrelevant sources or, conversely, excluding reliable or relevant sources. To mitigate this threat, we continued coding sources until theoretical saturation was reached, thereby reducing the impact of any potential outliers.

Hagstrom et al. [114] and Piasecki et al. [115] describe the use of Web search engines for finding sources when conducting literature reviews. Any search-based bias was mitigated by consistently applying these search methods and inclusion-exclusion criteria and by allowing our findings to guide us.

Grounded Theory aims to discover and explain phenomena that exist – it neither claims to exhaustively capture all such phenomena, nor does it claim to quantify their frequency or guarantee a specific distribution or coverage [116]. We similarly make no such claims, nor do we claim to document best practices or universally applicable recommendations, since the modelled practices and decisions are merely observed phenomena.

Considering every possible source would have been infeasible and would likely have brought little additional benefit, as we reached theoretical saturation. For the same reason, it is unlikely that a multivocal literature review would have led to dramatically different results. Thus, we are confident that our results accurately model practitioner understanding of the domain.

3.6 Conclusion

This Grounded Theory-based grey literature study modelled ADDs, decision options, relations and decision drivers in software engineering and software architecture for ML within the context of deployment. Our findings uncovered several significant, interrelated design decisions when deploying ML. We identified decisions involving *automated integration and delivery*, *pipeline triggering*, *model version deployment*, *delivery pipeline component tasks*, *testing within a pipeline*, *data versioning* and *MLOps*, with each decision being associated with various options that may be selected. The resulting model can be used to evaluate the specific decision drivers for each option of a given decision, guiding practitioners and advancing scientific understanding of typical practices. Potential applications of our ADDs include reducing uncertainty or complexity in the ML decision space, assessing conformance to established architectural practices or identifying anti-patterns in existing systems. Our work serves as the foundation for ongoing research in this field.

Part III

Architectural Knowledge for Reinforcement Learning and RLOps

4 Supporting Architectural Decision-Making on Training Strategies in Reinforcement Learning Architectures

In the dynamic landscape of AI, RL has emerged as a powerful paradigm for training intelligent agents to make sequential decisions. As RL architectures become increasingly complex, informed decision-making about training strategies and their related consequences on the software architecture becomes more intricate. This work addresses this challenge by presenting the outcomes of a qualitative, in-depth study focused on best practices and patterns in training strategies for RL architectures, as articulated by practitioners. Leveraging Straussian Grounded Theory and pattern mining, we introduce a formal architectural decision model to bridge the gap between scientific insights and practical implementation. We aim to enhance the understanding of practitioners' approaches to RL architecture. The chapter analyses 33 knowledge sources to discern established industrial practices, patterns, relations and decision drivers. Based on this knowledge, we introduce a formal ADD model, which encapsulates six decisions, 29 decision options and 19 decision drivers, providing robust decision-making support for this critical facet of RL-based software architectures.

4.1 Introduction

In the fast-evolving realm of AI, RL has emerged as a potent paradigm, enabling the training of intelligent agents to make sequential decisions [2], [3]. As RL architectures grow in complexity and scale, making pivotal decisions about training strategies and their consequences for software architecture becomes increasingly involved. Several authors have attempted to capture patterns and best practices in training strategies within RL architectures [47], [117], [118], [119]. However, these works predominantly concentrate on applying published patterns or scientific findings. In contrast, we find prevalent industry practices mostly in grey literature, such as blogs, experience reports and system documentation. Whilst these sources offer insights into existing practices, they often need more systematic architectural guidance. The reported practices exhibit variation and reliance on personal experiences, contributing to uncertainty and risk in RL design, and necessitating extensive experience or a detailed study of knowledge sources. We aim to present a more comprehensive and consistent perspective on current industrial practices, complementing existing knowledge.

To achieve this, we conducted an in-depth qualitative study of RL descriptions provided by practitioners, extracting informal information about established practices and patterns in RL training strategies. Employing Straussian Grounded Theory [76], [96] and pattern mining [120], we systematically analysed practitioner sources using coding and constant comparison methods, followed by precise software modelling. This rigorous process enabled the development of a detailed software model illustrating established practices, patterns and their interrelationships.

This chapter is based on **P3**, describes **RC3** and aims to study the following research questions as part of **RQ2** and **RP2**:

RQ2.1 *Which patterns and practices do practitioners currently use for training strategies in RL architectures?*

RQ2.2 *What are the relations between existing training patterns and practices? Specifically, which ADDs are pertinent when designing RL systems?*

RQ2.3 *What are the influencing factors (i.e. decision drivers) in architecting RL systems in the eyes of the practitioner today?*

We make two primary contributions in this chapter. Firstly, we conduct a qualitative study on RL architectures, analysing 33 knowledge sources to discern established industrial practices, patterns, relationships and decision drivers. Secondly, we develop a formal ADD model, which encapsulates six decisions, 29 decision options and 19 decision drivers.

This chapter is organised as follows: in Section 4.2, we describe RL and its role in software architecture. Section 4.3 compares our work with related studies. Section 4.4 presents the research method applied in our study and summarises the knowledge sources. Section 4.5 details our reusable ADD model on RL architectures. Section 4.6 evaluates and Section 4.7 discusses our results. Subsequently, Section 4.8 considers potential threats to the validity of our study and Section 4.9 summarises our findings.

4.2 Background: RL and its Role in Software Architecture

RL [2], a paradigm for training intelligent agents via trial-and-error in sequential decision-making, diverges from supervised learning by enabling agents to learn optimal strategies through interactions with an environment and by receiving feedback in the form of rewards or penalties. This process involves formulating the problem, defining state and action spaces and establishing a reward structure. Environmental design is crucial, as it shapes decision-making, actions and feedback. The RL training pipeline encompasses environment setup, policy definition, algorithmic training, hyperparameter tuning and policy evaluation, emphasising an iterative approach to continuous performance enhancement through parameter adjustments and repeated training.

Incorporating RL into software architecture is pivotal, as RL training strategy decisions greatly influence system design. For instance, choosing between single-agent and multi-agent RL models impacts system design by shaping communication channels and

coordination mechanisms. Similarly, the use of model checkpoints and multiple model versions affects architecture, requiring a checkpoint management system that influences storage, retrieval and deployment processes.

4.3 Related Work

In this section, we provide details on related works and compare them. We discuss related studies for RL patterns, practices and approaches for decision documentation. Several approaches that study RL patterns and practices exist.

Lee et al. [117] discuss the evolution of RL algorithms, progressing from single-agent to multi-agent systems. Their focus is on a distributed optimisation perspective.

Canese et al. [118] delineate multi-agent algorithms and compare them based on critical characteristics for multi-agent RL applications, including non-stationarity, scalability and observability. Their work also encompasses the most prevalent benchmark environments utilised to assess the performance of the discussed methods.

Zhu et al. [47] present a framework for classifying state-of-the-art transfer learning approaches, through which they scrutinise objectives, methodologies, compatibility with RL backbones and practical applications. Additionally, they establish links between transfer learning and other pertinent topics from an RL perspective, delving into potential challenges that await further research.

Eimer et al. [121] propose adopting best practices from AutoML in RL, including separating tuning and testing seeds and employing principled hyperparameter optimisation across a broad search space. Whilst these works focus on specific aspects of RL, our work contributes by conducting a comprehensive qualitative study to identify best practices and patterns in RL training strategies. Furthermore, the formal ADD model introduced in our work provides a structured framework and a focus on software architecture beyond these related works.

Washizaki et al. [122] present a comprehensive literature review that identifies 15 software engineering design patterns tailored to ML applications. The work by Sharma and Bavuluri [123] involves the identification and analysis of design patterns and architectural patterns in two software applications utilising ML and DL techniques, respectively. Whilst these works focus on design patterns in ML applications, our study specifically addresses RL architectures, which are not yet considered as design patterns or ADDs.

Furthermore, numerous approaches exist for decision documentation, ranging from service-oriented solutions [124] and service-based platform integration [125] to considerations of REST vs SOAP [126] and big data repositories [127]. However, these studies are not focused on training strategies in RL architectures or similar topics. Whilst other authors have combined decision models with formal view models [128], our contribution builds on these techniques by incorporating a formal modelling approach rooted in a qualitative research methodology.

In prior work [23], [24] described in this doctoral thesis, we employed a Straussian Grounded Theory-based approach to examine practitioners' current understanding and architectural concepts of ML solution deployment, formulating ADDs, decision options,

decision drivers and various relations for the ML workflow and ML deployment. Similarly, this study scrutinises practitioners' methods and techniques, aiming to bridge the gap between theory and practice in RL training strategies. The resulting formal model encompasses ADDs, decision options, practices, decision drivers and relations, providing valuable insights to help practitioners make informed decisions about RL training strategies.

4.4 Research Method

This section outlines the research method employed in this study and the modelling tool used to create and visualise the decision model.

4.4.1 Grounded Theory

This work aims to study established training strategies for RL architectures systematically. We follow the model-based qualitative research method described in [129]. It is based on the established Straussian Grounded Theory [76], [96] research method, combined with methods for studying established practices, such as pattern mining [120] and their integration with Grounded Theory [130]. It involves iterative steps for interpreting data, aiming to construct a theory rooted in the collected data. Data analysis is conducted during data collection rather than after.

Constant comparison is vital in Grounded Theory, where researchers continuously compare existing data and concepts with new data to identify emerging abstract concepts. These new concepts are then compared with pre-existing ones. We organised concepts into categories (or codes) and linked them with properties and relations, guiding subsequent research iterations.

In iterative cycles, we conducted knowledge-mining procedures, searching for new sources and applying open and axial coding to identify candidate categories. Continuously comparing codes with the evolving model enabled incremental enhancements. The decision of when to conclude this process is critical in qualitative methods, with theoretical saturation [76] widely accepted as a stopping criterion. In our study, we ceased analysis when an additional seven knowledge sources failed to contribute new insights, demonstrating theoretical saturation. Although this approach was conservative, our study had already achieved convergence after 26 knowledge sources.

Source selection, detailed in Table 4.1, drew upon our experience with tools, methods, patterns and practices encountered or studied previously. Our methodology encompassed three coding activities: (1) open coding involved developing concepts from data with specific questioning, precise coding and minimal assumptions; (2) axial coding focused on developing categories and linking data, concepts, categories and properties; (3) selective coding aimed to integrate developed categories into a core category.

Table 4.1: List of Knowledge Sources Included in the Study

ID	Description	Archive URL
S1	What is Model-Based Reinforcement Learning?	https://bit.ly/3MNg3iR
S2	Reinforcement Learning Meets Large Language Models (LLMs): Aligning Human Preferences in LLMs	https://bit.ly/47EbKP3
S3	What Is Better: One General Model or Many Specialised Models?	https://bit.ly/47gHtpJ
S4	Multi-Agent Reinforcement Learning with Coordination Graphs	https://bit.ly/3sAHkhA
S5	Model-Based RL for Decentralised Multi-agent Navigation	https://bit.ly/3sF42VV
S6	EFlow — Racing towards millions of ML flows	https://bit.ly/49BARnj
S7	L33T M10y	https://sforce.co/3MNfKom
S8	Generalists vs (Micro)Specialists in AI architectures	https://bit.ly/49FxXhu
S9	Hierarchical Reinforcement Learning for Robustness, Performance and Explainability	https://bit.ly/40CLNx3
S10	Hierarchical Reinforcement Learning	https://bit.ly/3R17wLO
S11	Single-agent vs multi-agent system in AI	https://bit.ly/40DwcNJ
S12	Train Agents Using Parallel Computing and GPUs	https://bit.ly/49GQNVi
S13	Centralised Training and Decentralised Execution in Multi-Agent Reinforcement Learning	https://shorturl.at/fpEN1
S14	Multi-Agent Reinforcement Learning (MARL) and Cooperative AI	https://shorturl.at/bciM3
S15	Decentralised Multi-Agent Reinforcement Learning and Game Theory	https://shorturl.at/fANQW
S16	What is Hierarchical Reinforcement Learning?	https://shorturl.at/tGVW1
S17	The Promise of Hierarchical Reinforcement Learning	https://shorturl.at/ekBN1
S18	Saving and Loading your RL Algorithms and Policies	https://t.ly/8BaaS
S19	Running RLlib Experiments	https://t.ly/DOnRn
S20	Accelerate Training in RL Using Distributed Reinforcement Learning Architectures	https://t.ly/JmhJZ
S21	Parallelism Strategies for Distributed Training	https://t.ly/Y9P7Q
S22	Deep Reinforcement Learning and Hyperparameter Tuning	https://t.ly/FQe2V

Continued on next page...

ID	Description	Archive URL
S23	Hyperparameter Tuning in Reinforcement Learning is Easy, Actually	https://rb.gy/3kj2hq
S24	Transfer Learning in Reinforcement Learning	https://rb.gy/xi6q56
S25	RL — Transfer Learning	https://rb.gy/a37rhn
S26	Introduction to Experience Replay for Off-Policy Deep Reinforcement Learning	https://rb.gy/dh0l5d
S27	Understanding Gradient Clipping (and How It Can Fix Exploding Gradients Problem)	https://rb.gy/2rdzuk
S28	How to Improve Your Network Performance by Using Curriculum Learning	https://rb.gy/hnhh4d
S29	Curriculum Learning	https://rb.gy/wgjs6p
S30	Parameter Sharing and Tying	https://rb.gy/enucvc
S31	A Complete Guide to Data Augmentation	https://rb.gy/so60h0
S32	Model Compression Techniques for Edge AI	https://rb.gy/84ih5l
S33	Distributed training with TensorFlow	https://urlis.net/myucb96s

4.4.2 Methodology

In Figure 4.1, we present an overview of our research method steps followed in this study.

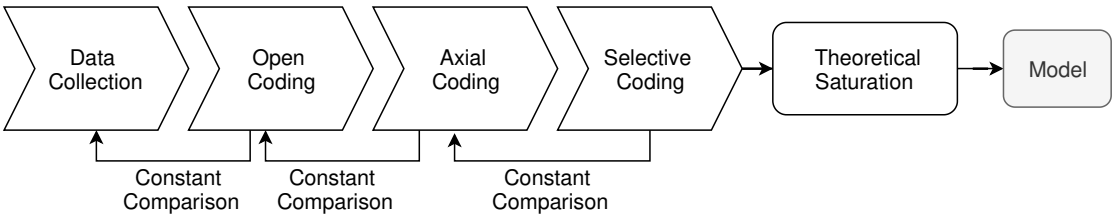

Figure 4.1: Research method steps.

To gather practitioner sources, we utilised standard search engines like Google, StartPage, Bing and DuckDuckGo, as well as topic portals such as InfoQ and DZone. We utilised relevant initial search terms to align with our work's focus, such as "training strategies in RL", "transfer learning in RL" and "checkpoints in RL". Our initial data came from these search terms. We opted against a multivocal literature review that includes scientific sources, as our research goals focus on practitioner views.

After the coding process, additional data sources emerged, including new keywords and new referenced sources. The data collection process was repeated using scanning and skimming techniques. During open and axial coding, we studied each included source line

by line in-depth in multiple iterations. We chose this method over manually browsing selected grey literature initially because it is replicable [113]. We employed Grounded Theory coding practices and constant iterative comparison to identify concepts, categories, properties and relations. We developed our decision model using our Python-based modelling tool, Codeable Models [78].

For subsequent iterations, we conducted searches using relevant terms derived from the identified topics, focusing on areas requiring coding and their potential contributions to the model. Practitioner articles needed to apply to the topic to be considered candidate sources rather than primarily promotional. We reviewed and approved each other's selection of sources for suitability.

Open coding was utilised to translate conceptual details into conceptual labels, whilst axial coding facilitated the identification of categories based on recurring, synonymous and related concepts. Each source underwent line-by-line examination during both open and axial coding phases, with our thought processes, conceptual understanding, interpretation and reasoning documented in memos linked to the respective sources, ensuring the traceability of codes to their origins.

Selective coding then focused on extracting the main ideas of the theory, refining previous sources as necessary. Formal modelling was employed for axial and selective coding, resulting in a precise and consistent theory represented as a UML model. All relevant lines in the knowledge sources were coded, providing an initial interpretation through open coding, followed by categorisation and formalisation via axial coding.

Python source code models[1] were generated from the information derived from the sources, facilitating further analysis and refinement. This iterative process culminated in identifying errors and achieving performance improvements during selective coding, thereby ensuring the accuracy and robustness of our findings.

4.5 Reusable ADD Model for Training Strategies in RL Architectures

In this section, we introduce the reusable ADD model derived in our study[2]. Figure 4.2 shows the metamodel for ADD models. The metamodel contains the *Decisions* of the ADD model. The *Decision* has a *Context*, which is described by a domain object that denotes the system part or aspect to which the *Decision* applies. Each *Decision* has *Options*. All *Options* are *Solutions*. An *Option* has *Forces*, which can have a *Force* impact. Finally, *Decisions* and *Options* (*Solutions*) can have *Relations*. A *Solution* can relate to another *Solution*, but the *Relation's* source or target must be an *Option* (relations such as *is-a*, *uses*, *can be combined with*, *typically realised with*). All *Solutions* in the *Decision* must be linked directly or via other *Options* to a *Decision*. *Decisions* and *Options* can also have *next-decision* relations. *Decisions*, *Contexts*, *Solutions* (and

[1] See the `Data/Decisions/Generated Python Models from Sources` directory in our replication package [131].

[2] See our replication package [131] for all study artefacts.

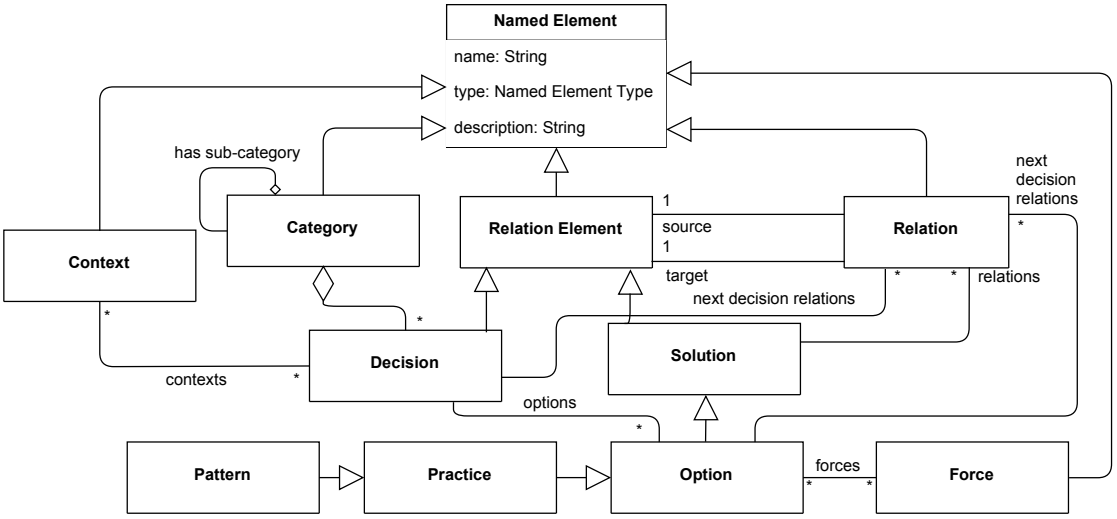

Figure 4.2: Metamodel for ADD models.

Options), *Forces*, and *Relations* are all *Named Elements*, meaning they can have an optional name and an optional description.

The reusable ADD model consists of a single *Decision Category* and the *Training Efficiency and Optimisation Strategies Category*, which has six top-level decisions, as depicted in Figure 4.3. A *Category* can have one or more decisions. All elements of our model are instances of the metamodel, and metaclasses include *Decision* and *Category*, amongst others. We denote the metaclasses as stereotypes in the model descriptions below.

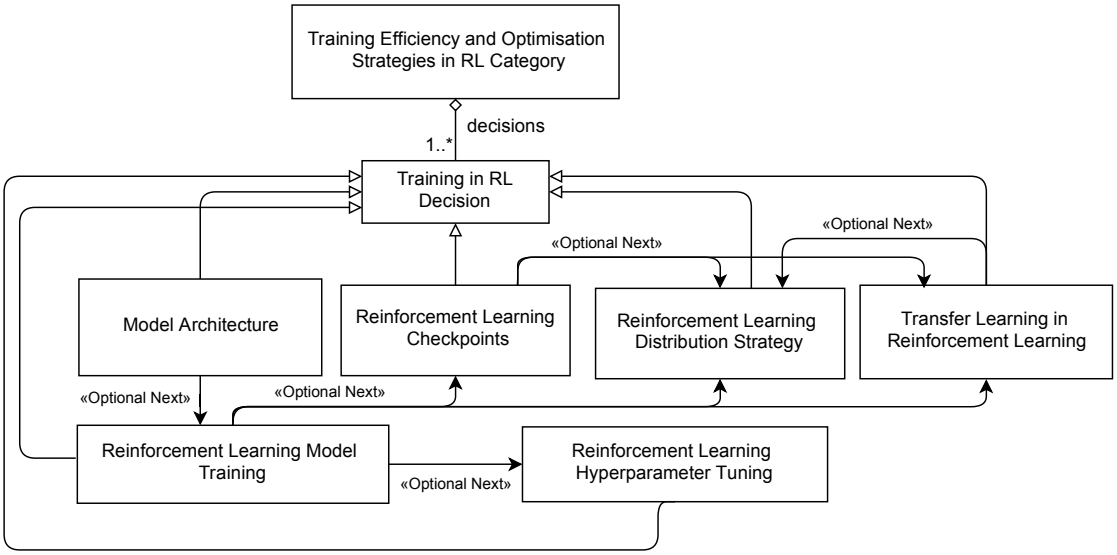

Figure 4.3: Overview of the reusable ADD model on training strategies in RL architectures.

4.5.1 Model Architecture

RL model architectures are pivotal in RL-based software architectures. There are various approaches to model architecture (Figure 4.4) [S1-S8, S10, S16, S17], each with advantages and trade-offs.

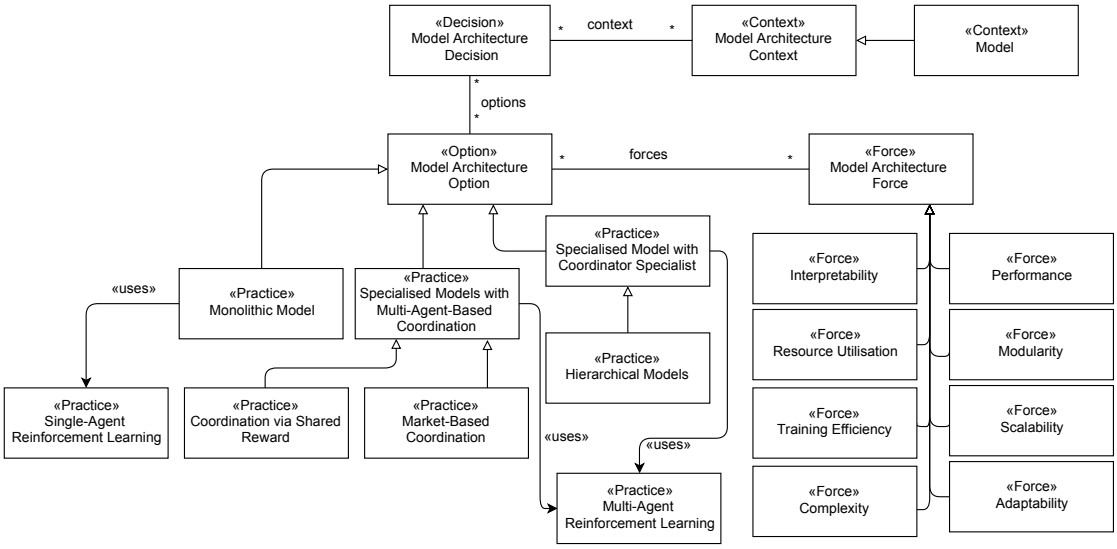

Figure 4.4: Model architecture decision.

One prominent option is the **Monolithic Model** where a single, comprehensive RL model is trained centrally to make all predictions. This approach is akin to having a centralised decision-maker that handles all aspects of an operation. This model can effectively capture complex interdependencies, ensuring a holistic understanding of the environment [2]. This pattern uses **Single-Agent RL**.

Specialised Models with Multi-Agent-Based Coordination present an alternative perspective, using **Multi-Agent RL**. This pattern involves multiple independent agents that collaborate to achieve a common goal. Coordination can be facilitated through shared rewards or market-based systems, allowing for decentralised decision-making and resource allocation [132].

The Specialised Models with a Coordinator Specialist introduce a hybrid solution that combines the benefits of specialised agents with the oversight of a central coordinator. This coordinator specialist, often an AI system, possesses a broader view of the entire process and makes high-level decisions based on information collected from specialised models. This architecture aims to strike a balance between distributed coordination and centralised control [3]. This pattern uses **Multi-Agent RL** for training specialised models.

Hierarchical Models offer another dimension to RL architectures, featuring multiple levels of control. A top-level controller manages the broader process, whilst lower-level controllers handle specific sections or processes. This approach provides a structured framework for decision-making, enabling more efficient handling of complex systems [132].

The **Monolithic Model** gains strength from its lower *complexity* as it tackles the intricate challenges of building and maintaining a comprehensive RL system. However, a **Monolithic Model's** *flexibility* may be limited, especially in dynamic environments requiring rapid adjustments. Adapting to changes may be more challenging due to the model's overarching nature. **Specialised Models with Multi-Agent-Based Coordination** can enhance the *performance* and *adaptability* with which specialised models can excel at their designated tasks, leading to efficient overall system performance and more efficient *resource utilisation*. In contrast, the **Specialised Models with a Coordinator Specialist** can also enhance *performance*. Specialised and hierarchical models may offer better *scalability*, as they can be expanded or adapted to support *modularity*. The **Monolithic Model** may offer better *interpretability* by providing a unified structure, making it easier to understand and interpret the decision-making process. Furthermore, it may improve *training efficiency*, as it requires less training time and computational resources than more complex architectures.

4.5.2 RL Model Training

There are several patterns and practices related to RL training strategies (Figure 4.5) [S4-S21].

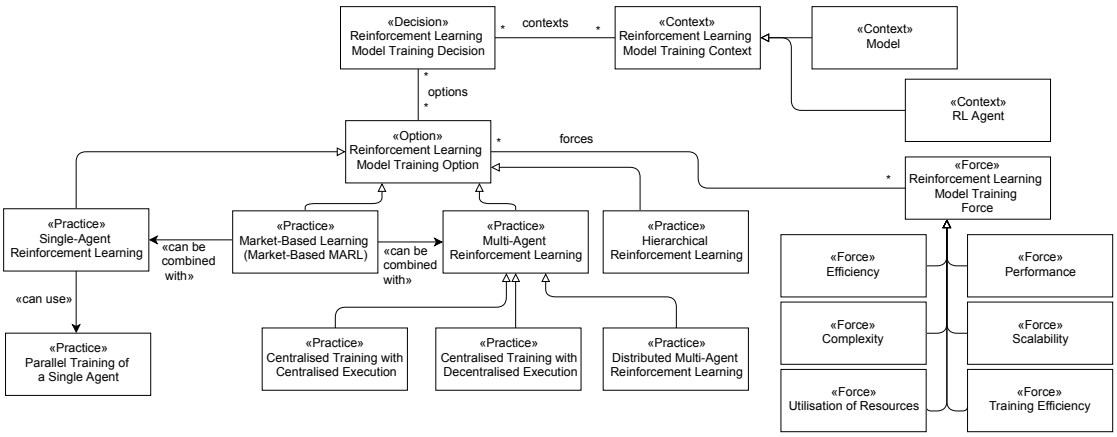

Figure 4.5: RL model training decision.

Single-Agent RL is a viable strategy for employing a monolithic process control model. In this scenario, a single model undertakes the formidable task of learning to manage all facets of production control. A specialised technique, **Parallel Training of a Single Agent**, is employed to expedite learning without introducing multiple agents interacting. This involves running multiple instances of the same task in parallel, a process known for its efficiency gains in RL [133]. On the other hand, the utilisation of **Multi-Agent RL (MARL)** aligns with model architectures involving specialised models with distributed coordination.

The training methods within MARL encompass distinctive strategies, such as **Centralised Training with Centralised Execution**, in which agents are trained with access to

a centralised controller that provides complete environmental state information. This centralisation extends to the execution phase, ensuring consistent access to global information. Alternatively, **Centralised Training with Decentralised Execution** allows agents to train with centralised information but execute decisions independently. Additionally, the **Distributed MARL** framework involves multiple agents trained independently on different computational nodes or devices that collaborate or compete in a shared environment.

Another practice in RL training strategies is **Market-Based Learning**, in which agents operate in a market-like environment, exchanging resources or making decisions based on market dynamics. This practice can be combined with **Single-Agent RL** and **Multi-Agent RL** and often involves mechanisms such as auctions, negotiations or trading. Another pattern, **Hierarchical RL**, introduces a hierarchical decision-making process in which high-level controllers set goals and lower-level controllers execute actions based on them [132].

Single-Agent RL, particularly with parallel training, benefits from parallelisation techniques to improve *training efficiency*, but may face challenges in highly complex systems. As a structured approach, **Hierarchical RL** enhances *scalability* and mitigates *complexity* by allowing agents to focus on localised adaptations within a hierarchical framework. **Parallel Training of a Single Agent** enhances *training efficiency* and *performance* by leveraging parallelisation techniques.

4.5.3 RL Checkpoints

Checkpoints (Figure 4.6) [S18, S19, S26] are a pivotal practice in RL model development. They serve a multifaceted purpose, primarily enabling resumable training by saving the model's parameters, optimiser state and related data at specific intervals. This ability to resume ensures that in the event of an interruption or system failure, training can be seamlessly continued from the last checkpoint, promoting *robustness* in the training process. Furthermore, **Checkpoints** play a crucial role in version control, capturing the model's evolution at different stages of development. In collaborative settings, this allows for easy tracking of changes and facilitates comparison between different versions. Additionally, **Checkpoints** are instrumental for implementing early stopping strategies, in which the model's progress is periodically evaluated on a validation set. If the model fails to improve after a set number of epochs, training can be halted, and the model reverted to the checkpoint corresponding to its peak performance. This practice helps prevent overfitting and ensures the development of a well-generalised RL model. The decision has two options: **No Use of Checkpoints** and **Use of Checkpoints**.

Whilst this pattern enhances the *robustness* and *efficiency* of workflows, it introduces considerations such as storage and *computation overheads*, as well as *complexity* in implementation. Despite these drawbacks, the benefits of preserving the model state far outweigh the disadvantages, ensuring a more *reliable* and *flexible* training process.

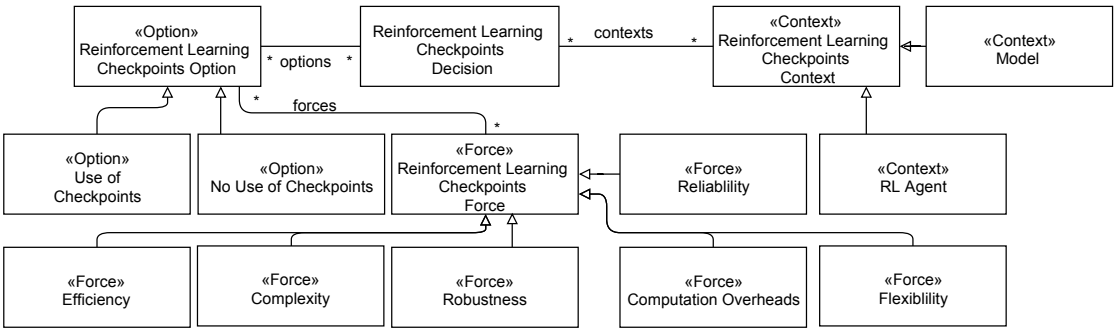

Figure 4.6: RL checkpoints decision.

4.5.4 Transfer Learning in RL

Transfer Learning [47] (Figure 4.7) [S24, S25, S26] involves leveraging knowledge gained from solving one problem and applying it to a different but related problem. In RL, transfer learning is a powerful tool for handling new tasks or environments. The process typically involves taking a pre-trained model, often trained on a large, diverse dataset or on a different but related task and adapting it to the target RL problem.

Transfer Learning is particularly beneficial when the source task shares some underlying features or patterns with the target task. It allows the RL agent to leverage relevant knowledge from the source task, providing a head start in learning the intricacies of the new environment. This approach is especially valuable in situations where collecting large amounts of task-specific data is expensive or time-consuming. This decision includes two main options: **No Use of Transfer Learning** and **Use of Transfer Learning**.

The use of transfer learning offers several advantages that enhance *efficiency* and *adaptability*. One key benefit is *data efficiency*, where transfer learning leverages knowledge from a pre-trained model trained on a source task with abundant data. This knowledge encompasses learned feature representations, reducing the reliance on extensive amounts of task-specific data. *Resource efficiency* is another compelling advantage, as transfer learning allows the reuse of pre-trained model weights rather than training a model from scratch, significantly reducing *computational resources*, leading to faster model convergence and lower hardware requirements. The application of transfer learning also results in *improved generalisation*, as the model extracts high-level features and representations from the source domain, thereby enhancing its capacity to generalise to new tasks, even with limited target data. Additionally, transfer learning facilitates domain adaptation by enabling the model to adapt to shifts in data distribution. The model identifies key features and knowledge from the source domain and effectively applies them to the target domain. These advantages make transfer learning particularly valuable in scenarios with limited data and computational resources, thereby enhancing the model's *adaptability* and *generalisation* capabilities.

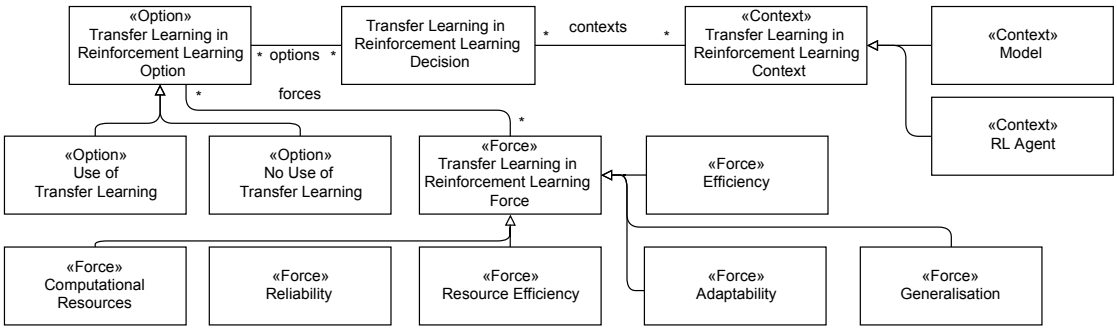

Figure 4.7: Transfer learning in RL decision.

4.5.5 RL Distribution Strategy

In RL, a resilient **Distribution Strategy** (Figure 4.8) [S12-S15, S20, S30, S33] is vital for efficiently managing computationally demanding tasks [134].

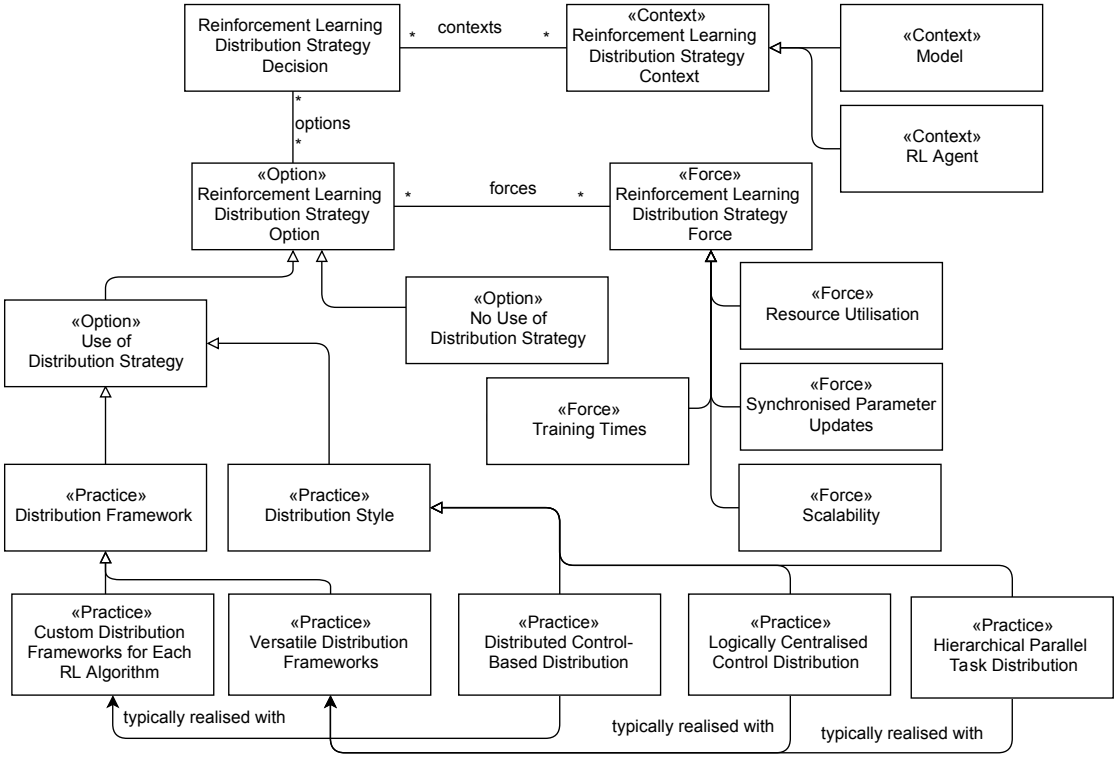

Figure 4.8: RL distribution strategy decision.

Distributed RL incorporates strategies like parameter server architectures and distributed experience replay [119]. Actor-critic architectures, exemplified by A3C [135] and Ape-X, facilitate effective learning by combining independent exploration with a central value function.

Parameter-sharing architectures, as seen in *Distributed Distributional Deterministic Policy Gradients* (D4PG) and policy gradient methods like *Proximal Policy Optimization* (PPO) [136] and *Trust Region Policy Optimization* (TRPO) [137] can be parallelised for faster training. The selection amongst these approaches hinges on factors such as task complexity and available computational resources [134]. This decision includes two options, namely **No Use of Distribution Strategy** and **Use of Distribution Strategy**.

From a software architecture perspective, developers often contend with a variety of frameworks, such as parameter servers, Message Passing Interface (MPI)-like collective communication primitives and task queues, to implement RL algorithms effectively [119]. As algorithms become more complex, there is a tendency to build custom distributed systems in which processes operate autonomously and coordinate without centralised control. This necessity arises from the irregular computational nature of RL algorithms, which challenges conventional distribution frameworks and prompts developers to integrate various solutions for efficient implementation.

Consequently, a choice emerges between **Custom Distribution Frameworks for Each RL Algorithm** and **Versatile Distribution Frameworks**, like Ray/RLlib and TensorFlow [138], which offer abstraction across a wide range of RL algorithms. Most RL algorithms today use a fully distributed style [119].

Distributed Control-Based Distribution involves implementing RL algorithms across independent, replicated processes that coordinate via various mechanisms. Such coordination is suitable for algorithms with minimal interaction and independent components.

Logically Centralised Control Distribution employs a central controller delegating tasks to parallel processes, simplifying implementation. It suits scenarios where coordination is critical, and a single control point can manage parallel execution effectively.

Hierarchical Parallel Task Distribution extends the logically centralised model, allowing nested delegation for complex algorithms. It enhances flexibility and scalability, making it suitable for tasks with varying levels of parallelism and complex dependencies.

Relating to the choice between **Custom Distribution Frameworks for Each RL Algorithm** and **Versatile Distribution Frameworks**, **Distributed Control-Based Distribution** aligns with custom frameworks, offering precision for specific algorithm needs [119].

Logically Centralised Control Distribution and **Hierarchical Parallel Task Distribution** are usually employed by versatile frameworks, providing a unified solution for diverse RL algorithms [119].

Use of Distribution Strategy further enhances *resource utilisation* by dividing the training dataset amongst different machines, enabling simultaneous processing and aggregation of updates. To coordinate this distributed effort, sophisticated distributed optimisation algorithms ensure *synchronised parameter updates*, preventing conflicts and promoting convergence.

The combination of these strategies yields accelerated *training times* and *scalability*, making it particularly beneficial for handling large datasets and complex RL models. This distributed approach represents a significant paradigm shift in the training of RL

models, addressing the computational challenges inherent in developing sophisticated and high-performance RL systems.

4.5.6 RL Hyperparameter Tuning

RL Hyperparameter Tuning [121] (Figure 4.9) [S19-S23] is a critical aspect of optimising the performance and efficiency of RL models.

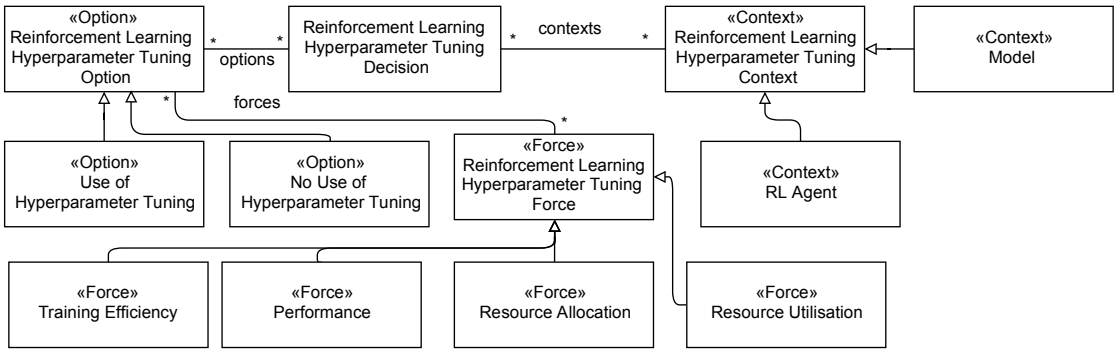

Figure 4.9: RL hyperparameter tuning decision.

Hyperparameters are external settings that control the behaviour of the learning algorithm, and finding the right combination is crucial for achieving optimal results. This decision includes two options: **No Use of Hyperparameter Tuning** and **Use of Hyperparameter Tuning**.

Optimising hyperparameters in RL models yields multifaceted benefits. The quest for optimal hyperparameters directly translates into improved model *performance*, amplifying the learning capabilities of RL models and improving *training efficiency*. Additionally, the systematic exploration of the hyperparameter space contributes to efficient *resource utilisation*. By avoiding computational expenditure tied to suboptimal configurations, these methods ensure that computing *resources are allocated* judiciously, optimising both time and energy. Furthermore, the time savings afforded by automated hyperparameter tuning are a crucial advantage. By automating the tuning process, researchers and practitioners can reallocate their time and effort to other critical facets of model development, fostering a more streamlined, productive workflow.

4.6 Evaluation

We systematically developed an ADD model by closely aligning with the selected sources as outlined in Table 4.1. The nomenclature for the ADD model elements was derived from the terminology employed in these sources, and we established generic type names based on these element names. When introducing a new type name, we conducted a thorough comparison with existing names to ascertain its necessity. As depicted in Figure 4.10, the point of theoretical saturation in Grounded Theory [76] was reached after

assimilating twenty-six sources. Across the initial thirteen sources, frequent adjustments to designated type names were necessary. However, in the subsequent thirteen sources, such modifications occurred less frequently. Notably, no further alterations were required for the remaining sources.

We used the counts of new model element types and relation types introduced per source as indicators of theoretical saturation. This method is a reasonable approach, as theoretical saturation occurs when the researcher can no longer uncover new concepts or relationships in the data. In our models, the introduction of new model elements and relations signifies the emergence of novel concepts and relationships.

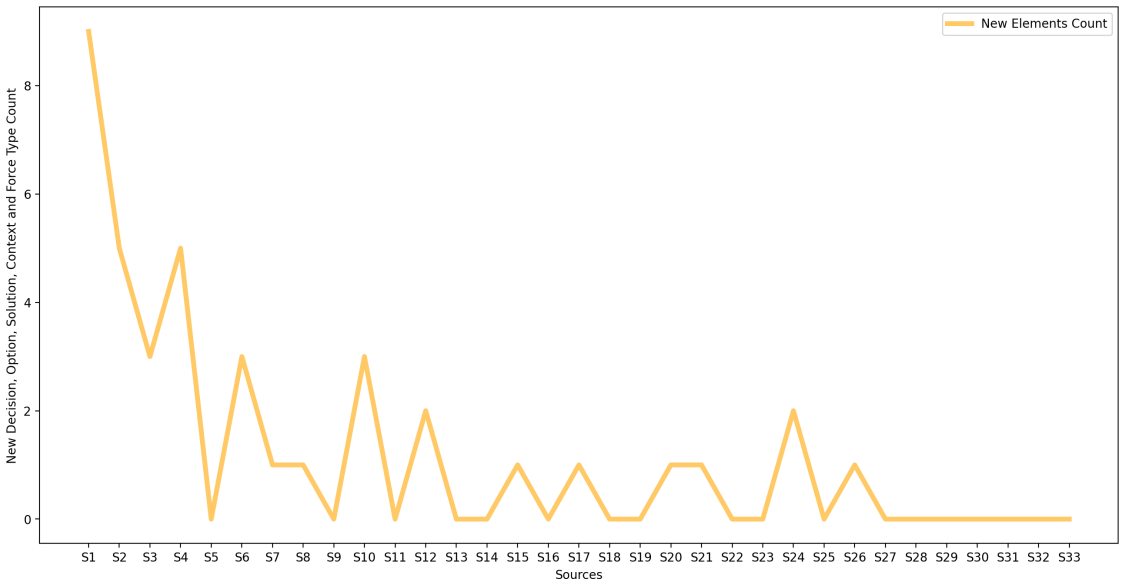

Figure 4.10: Number of elements of newly-added sources.

4.7 Discussion

This section discusses our findings for the research questions from Section 4.1.

RQ$_{2.1}$ *Which patterns and practices do practitioners currently use for training strategies in RL architectures?* After analysing 33 practitioner knowledge sources, we discovered evidence for 29 patterns and practices currently used by practitioners for supporting training in RL architectures, which we modelled as ADD decision options. These patterns and practices are associated with ADDs and were found to be independent of each other. Exceptions are the **Monolithic Model** and **Specialised Models with Multi-Agent-Based Coordination** patterns, which can use the **Single-Agent RL** practice. Another commonality is that the **Specialised Models with Multi-Agent-Based Coordination** and **Specialised Model with Coordinator Specialist** patterns are used for training the **Multi-Agent RL**. The adoption of **Specialised Models with Multi-Agent-Based Coordination** or **Hierarchical Models** is motivated by the pursuit of *adaptability* and

efficient resource utilisation, albeit at the cost of increased *complexity*. These strategies reflect the nuanced decision-making landscape where practitioners balance architectural choices with the specific demands of their applications.

Moreover, **Single-Agent RL** is applied to **Monolithic Model**, often with **Parallel Training of a Single Agent** for efficiency gains. **Multi-Agent RL** involves distinctive strategies, such as **Centralised Training with Centralised Execution** and **Centralised Training with Decentralised Execution**, as well as **Distributed MARL**, where agents train independently on different nodes. **Market-Based Learning** introduces market dynamics, whilst **Hierarchical RL** features a hierarchical decision-making process. Furthermore, the implementation of **Checkpoints** is crucial for resumable training, version control and early stopping strategies. The practice of **Transfer Learning** leverages knowledge from one task to another. A **Distribution Strategy** optimises RL model training using **Distributed Control-Based Distribution**, **Logically Centralised Control Distribution** and **Hierarchical Parallel Task Distribution**. **Hyperparameter Tuning** is essential for optimising model performance and training efficiency by systematically exploring the hyperparameter space.

RQ$_{2.2}$: *What are the relations between existing training patterns and practices? Specifically, which ADDs are pertinent when designing RL systems?* Given the central **Training Efficiency and Optimisation Strategies** category, we identified six top-level ADDs for supporting training in RL architectures. Our research revealed subtle relations between ADDs and decision options. For instance, **Specialised Models with Multi-Agent-Based Coordination** leverage **Multi-Agent RL** to enable decentralised decision-making through collaboration. **Single-Agent RL** is employed in a monolithic setting, where a single model manages all facets of production control and utilises **Parallel Training of a Single Agent** to expedite learning without introducing multiple interacting agents.

RQ$_{2.3}$: *What are the influencing factors (i.e. decision drivers) in architecting RL systems in the eyes of the practitioner today?* Through our research, we identified 19 influencing factors (forces) relevant to architecting training in RL architectures from the practitioners' perspective. Whilst these forces tended to be specific to individual ADDs and decision options, we also recognised some commonalities.

The choice between a **Monolithic Model**, **Specialised Models with Multi-Agent-Based Coordination**, and **Specialised Models with a Coordinator Specialist** depends on factors such as *complexity*, *flexibility*, *performance*, *adaptability*, *resource utilisation*, *scalability* and *interpretability*. Hierarchical models offer a structured framework that enhances decision-making efficiency. **Single-Agent RL** or **Multi-Agent RL** introduce further considerations related to *training efficiency*, *performance*, *scalability* and *modularity*. The incorporation of **Checkpoints** enhances both *robustness* and *efficiency*, although challenges such as storage and computation overheads must be considered. **Transfer Learning** and **Distribution Strategy** further influence *efficiency*, *adaptability* and *resource utilisation*. Additionally, **RL Hyperparameter Tuning** plays a critical role in optimising model *performance*, *training efficiency* and *resource utilisation*.

Given the crucial role of the aforementioned forces across various ADDs and their associated decision options, practitioners are advised to assess their significance early in

the architectural planning phase. This assessment can guide informed decision-making throughout the system's development.

4.8 Threats to Validity

In this section, we discuss threats to validity according to the threat types by Wohlin et al. [85].

To enhance internal validity, we included independent practitioner reports in our study rather than interviews, for instance, thereby minimising potential bias arising from participants' awareness during interviews. Whilst this approach mitigates bias, there is a risk of missing crucial information in reports that could have surfaced in interviews. To address this, we examined various sources extensively, exceeding the theoretical saturation requirements. Considering diverse sources minimises the chance of collectively overlooking vital information.

To mitigate researcher bias, different team members independently cross-verified all models. However, a potential threat to internal validity remains, as biases within the research team may persist and affect the modelling. Despite potential variations in modelling approaches by other researchers, our study's overarching goal of establishing a comprehensive model for all observed phenomena mitigates the significance of this threat.

We also acknowledge the potential bias introduced by the author team's experience and our search-based procedure for identifying knowledge sources. However, our research method, which relies on additional sources meeting inclusion and exclusion criteria rather than a specific distribution, largely mitigates this threat. This approach aligns with standard selection practices for interview participants in qualitative research studies in software engineering. Nevertheless, there is a risk of unintentionally excluding specific sources, which we addressed by forming an author team with extensive field experience and conducting thorough, broad searches. Despite the diverse range of included sources, our results likely generalise to various architectures that use training strategies. However, a threat to external validity persists, suggesting that our findings apply only to similar types of RL architectures.

We acknowledge the potential bias resulting from the restricted scope of our study's data and the absence of pertinent architectural solutions. Whilst our inclusion of numerous sources suggests potential generalisability to various architectures requiring training strategies, the threat to external validity persists. Our findings may only apply to similar training strategies in RL architectures, limiting their generalisation to novel or unconventional RL architectures without model modification.

4.9 Conclusion

We conducted a literature review using Grounded Theory to develop a model of training strategies for RL architectures, encompassing ADDs, decision options, relations and decision drivers. Our research centred on supporting training strategies in RL architectures and addressed three key research questions. In **RQ$_{2.1}$**, an analysis of 33 practitioner

knowledge sources revealed 29 options (patterns and practices) employed in RL training. These were modelled as ADD decision options that demonstrated a high degree of independence. **RQ$_{2.2}$** explored the top-level ADDs for training strategies in RL architectures, identifying six key ADDs and uncovering subtle relationships between them, thereby providing crucial insights for RL architectural planning. In **RQ$_{2.3}$**, we identified 19 influencing factors (forces) impacting training in RL architecture design, recognising their variations across individual ADDs and their options.

5 Bridging the Gap Between MLOps and RLOps: An Industry 4.0 Case Study on Architectural Design Decisions in Practice

In the domain of Industry 4.0 CPPSs, RL has gained momentum as an effective strategy for training intelligent agents via digital twins. Whilst the practice of MLOps has become established as a holistic approach to automating workflows in supervised and unsupervised ML, the extent to which MLOps practices apply to RL, particularly given major differences between ML and RL in model deployment and training, remains unclear. The literature on RLOps as a paradigm is scarce. We tackle this open question by conducting an exploratory, qualitative, deductive-inductive industry case study on a CPPS, performing content analysis of CPPS artefacts, such as architectural schematics and source code, and understanding their relation to 22 known ADDs and 86 associated decision options through classification into four distinct emergent categories. Our findings help bridge the gap between MLOps and RLOps architectures, providing novel insights into applying MLOps practices to RL and offering practical guidance and inspiration for further research.

5.1 Introduction

Industry 4.0 [10] has revolutionised manufacturing by integrating disparate digital technologies, leading to the development of CPPSs [15], which combine physical and computational elements to improve efficiency, flexibility, and customisation [13]. Within this setting, RL [2] has emerged as an effective technique for training intelligent agents with digital twins, allowing for real-time decision-making and process optimisation [16], [61]. MLOps has become a standard method for automating workflows in supervised and unsupervised ML, including data processing, model training, deployment and maintenance [4]. However, the application of MLOps concepts to RL – a burgeoning practice known as RLOps [7] – is largely unexplored. The potential applicability of MLOps to RL is unclear due to fundamental differences between ML and RL, for instance, in model deployment and training. Whilst ML models are usually trained on static datasets, RL models learn by interacting with dynamic environments, necessitating continual adaptation and real-time decision-making, with their own set of challenges [139].

The scarcity of scientific literature on RLOps creates a gap in our understanding of how to integrate RL into the MLOps paradigm successfully. Addressing this gap is critical for improving CPPS capabilities and realising the full promise of Industry 4.0 [11]. We intend

to bridge the knowledge gap between MLOps and RLOps by performing an exploratory, qualitative industry case study on a CPPS.

Our objectives are to evaluate the applicability of ML-specific ADDs to RL and RLOps, investigate the relationship between MLOps techniques and RL in a CPPS, and provide a reference for practitioners and a framework for future study. The results should add to the expanding body of knowledge regarding Industry 4.0, MLOps and RLOps, aiding practitioners in improving the efficiency and adaptability of CPPSs and providing fresh perspectives on integrating RL into MLOps.

We conducted an exploratory, qualitative industry case study of a CPPS. We used content analysis [67] to determine how 86 decision options derived from 22 ADDs related to 15 CPPS artefacts such as source code and architectural schematics, which we describe in more detail in Section 5.4.4.

This chapter is based on **P4**, describes **RC4** and addresses **RP2**. We aimed to answer the following research questions within **RQ2**:

RQ2.4 *To what extent are known ML and RL ADDs applicable to RL in a CPPS context?*

RQ2.5 *Which ADD decision options typically used in ML and RL apply directly to a real-world CPPS that utilises RL?*

RQ2.6 *Is it necessary to adapt ADD decision options typically used in ML for use with RL in a CPPS context?*

We make three main contributions in this chapter. Firstly, we conduct a qualitative industry case study on ML and RL practices in a real-world CPPS, modelling 15 system artefacts to establish the use of relevant ADDs, decision options, practices and patterns. Secondly, we interpret our findings and derive four categories of ADD decision options in an RL context. Finally, we identify ML practices that, at first sight, may not appear applicable to RL but can be easily adapted. In summary, we bridge the knowledge gap between MLOps and RLOps, providing practitioners with valuable insights and a reference point.

The rest of the chapter is structured as follows: in Section 5.2, we describe the concepts of ADDs, ML, RL, MLOps, Industry 4.0 and CPPSs. Section 5.3 compares our work with related studies. In Section 5.4, we describe the case study and our approach. Section 5.5 details our results, which we discuss in Section 5.6. We consider threats to validity in Section 5.7, and Section 5.8 concludes and suggests ideas for future work.

5.2 Background

5.2.1 ADDs

ADDs are a crucial aspect of software development that represent architectural decisions made during the design process [17], [18]. ADDs and their associated decision options encapsulate the rationales for choosing architectural elements, their relationships, and the trade-offs (in terms of *forces* or *decision drivers*) considered to meet functional and quality requirements [140].

In Sections 5.5 and 5.6, we list the ADDs and decision options of relevance to this study. A central ADD is whether to adopt MLOps as a holistic approach. Other ADDs are mostly concerned with data processing (e.g. *Automatic Data Processing*, *Data Ingestion* and *Data Versioning*), triggers (e.g. *Pipeline/Orchestrator Triggers*), model building (e.g. *Model Building Pipeline Tasks* and *Training Strategy*) and model deployment (e.g. *Delivery Pipeline Tasks* and *Model Version Deployment*).

Automation is represented as a high-level ADD (i.e. whether *AutoML* is used) and as a cross-cutting concern that can be considered in the context of decision options in practices related to data processing, pipeline/orchestrator triggers, pipeline tasks, model training, model building, model integration and delivery. The ADDs associated with RL are related exclusively to model architecture, training and distribution strategies. Specifically, the *Model Architecture* and *Model Training* variations specifying **single-agent** [141] vs **multi-agent** [142], [143] training, the use of *Checkpoints*, whether *Transfer Learning* is utilised, the *Distribution Strategy* adopted and the use of *RL Hyperparameter Tuning*.

5.2.2 ML, RL, MLOps and RLOps

ML and RL differ in their methodologies and applications. Whereas ML employs supervised or unsupervised learning techniques that learn from data to make predictions or classifications, RL focuses on learning optimal actions through interactions with an environment and feedback in the form of rewards or penalties for prior actions.

Given this distinction, one might assume the need for distinct operational frameworks for automating ML and RL. MLOps, already well-established, involves practices and tools for data processing and management, model training, deployment, management, and maintenance in production. RLOps, an RL-equivalent operational framework that has been sparsely documented in the literature, must address the unique challenges associated with the RL workflow and production environments, particularly differences in data processing, model architecture, training, distribution, and the complexities of real-time decision-making and dynamic environments.

5.2.3 Industry 4.0 and CPPSs

Industry 4.0 [10] represents a paradigm shift in manufacturing, driven by the integration of digital technologies such as the Internet of Things, AI and cloud computing, to create smart factories. Smart factories are interconnected systems that enable real-time data exchange and autonomous decision-making, enhancing efficiency, flexibility and customisation [11].

CPPSs, which merge physical (e.g. tools, robots and sensors) and computational (e.g. communication networks, data analytics and AI) elements, are central to Industry 4.0. CPPSs monitor, control and optimise manufacturing operations [16], [61]. Seamless interaction between the physical and digital realms using digital twins can improve production quality, reduce costs and simplify maintenance [144], [145].

5.3 Related Work

RLOps is sparsely mentioned in the scientific literature, though it is more commonly discussed in practitioner sources, such as code repositories, libraries, commercial platforms, and frameworks that support RLOps. This disparity suggests that RLOps is a relatively new field that is gaining traction in practice but exhibits a knowledge gap from a scientific perspective. Indeed, we could not find any Industry 4.0 case studies focusing on RLOps.

Li et al. [7] study RL as applied to open radio access networks (O-RAN). They highlight the differences between ML and RL, provide a taxonomy of ML/RL model lifecycle challenges, and suggest best practices for RLOps. They also design and implement a data analytics platform for O-RAN. As with our study, they recognise the differences between ML and RL and consider how MLOps can be applied to RL. Unlike our study, they focus on the model lifecycle rather than a wide range of ADDs, consider O-RAN rather than Industry 4.0 and do not conduct a case study.

Del Real Torres et al. [146] conduct a review of deep RL approaches for smart manufacturing in Industry 4.0 and 5.0 [147], noting the relevance of edge devices and cyber-physical systems. They consider a range of deep RL algorithms and their applicability in manufacturing processes, providing standard guidelines for developing and improving factories. Similarly to our study, they consider RL in the context of Industry 4.0. Unlike our study, they conduct a literature review rather than a case study, focusing on algorithms rather than MLOps and architectural aspects.

Kegyes et al. [148] conduct a systematic literature review of RL applications and methods used in Industry 4.0. As in our study, their work serves as an overview and reference. However, they focus on the theoretical underpinnings of RL and implementation details rather than architectural decisions and MLOps. Unlike our study, they surveyed the literature rather than conducting a case study.

Zhou et al. [31] recognise the need for MLOps as an application of DevOps [21], [22] principles to unify ML system development and operation, addressing the unique challenges faced when developing and deploying ML applications. They build an ML platform using DevOps and then measure its performance, providing a reference ML pipeline platform. Despite using a system as a case study, their example is not a case study of a real-world Industry 4.0 application and does not consider RL.

In contrast to the related work described above, our study examines the relevance of MLOps ADDs in an Industry 4.0 setting as applied to RL. It aims to bridge the gap between theory and practice in understanding how associated practices apply in that context. To our knowledge, ours is the first study to do so.

5.4 Case Study

This section provides a detailed description of the case study design and research process, which is depicted in Figure 5.1. Our research methodology adheres to the case study guidelines outlined by Runeson and Höst [88], and Yin [89].

The study was conducted within the framework of a previously established academia-

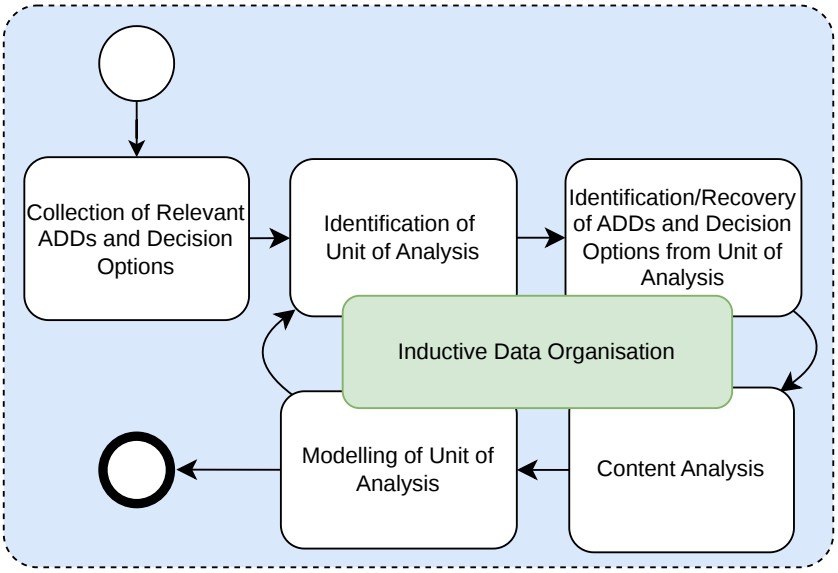

Figure 5.1: An overview of the research process we followed in this study.

industry project collaboration between the authors and Siemens Aktiengesellschaft Österreich, a large multinational technology company that provides production automation solutions. The project was chosen due to its relevance to both partners involved in the collaboration, and it focuses on an RL-based CPPS solution being developed by a team of experts at Siemens to automate factory manufacturing processes.

5.4.1 Research Process

The first step was the **Collection of Relevant ADDs and Decision Options**, which we sourced from our prior work [8], [23], [24] described earlier in this doctoral thesis, where we had derived them from practitioner grey literature using Straussian Grounded Theory [65]. The following four steps (collectively the **Inductive Data Organisation**) were performed iteratively until the entire *study object* (see Section 5.4.3) had been covered. The **Identification of Unit of Analysis** step involved finding the next potential *unit of analysis* (see Section 5.4.4). **Identification/Recovery of ADDs and Decision Options from Unit of Analysis** was a check to see if the current *unit of analysis* evidenced the use of a decision option. If so, **Content Analysis** of the *unit of analysis*, as described in Section 5.4.5, was performed. Finally, if the *unit of analysis* used a decision option, **Modelling of Unit of Analysis** (see Section 5.4.6) involved writing component, pipeline and process models in Python and generating their UML visualisations.

5.4.2 Approach

Our study employs an empirical, qualitative design, examining contemporary phenomena within their contexts [89]. Unlike action research [88], which aims to effect change, our

approach is observational, prioritising realism over control. We focus on studying and analysing project artefacts, and are interested in theory-building and in understanding how practitioners have applied ADDs. We aim to avoid influencing practitioners or introducing bias, for instance, via interviews, and we always strive to maintain a neutral frame of reference.

Despite the study's observational nature, practical uses of our findings are discussed in Sections 5.6 and 5.8. This exploratory study aims to investigate the implementation of the *study object* (see Section 5.4.3) to gain new insights and understanding, address our research questions, and generate ideas for future research.

Additionally, the study is descriptive, providing a detailed portrayal of a specific situation and its associated phenomena. The study design incorporates the flexibility to effectively manage "*the complex and dynamic characteristics of real-world phenomena, like software engineering*" [88].

5.4.3 Study Object

The focus of our research (the *study object*) is an innovative, proprietary software system developed by domain and RL experts within Siemens. This advanced solution integrates RL with a CPPS, leveraging MLOps and RLOps practices for an Industry 4.0 manufacturing environment. The system's task is to automate various aspects of industrial product manufacturing, including physical modification and assembly of components.

5.4.4 Units of Analysis and Data Collection

The *units of analysis* are the data sources within the *study object* related to ML, RL, MLOps, and RLOps. A range of disparate data sources enabled us to triangulate the phenomena we encountered. Specifically, we practised *third-degree data collection*, which involves the independent analysis of available work artefacts. We made use of project source code, CI/CD pipeline definitions and configuration files, high-level schematics, informal architectural diagrams, UML diagrams, presentations by team members and associated slide decks, and observation of team behaviour practices (e.g. source repository commits and their comments) to gain an understanding of the system and to model relevant parts.

Since we did not conduct formal interviews and this is not an ethnographic study in which, for instance, cultural aspects within an organisation are studied, there are no human subjects in that sense. However, informal discussions and meetings with partners from Siemens throughout the normal course of our project collaboration helped clarify and disambiguate our understanding of the system and practices.

5.4.5 Content Analysis

Using the ML and RL ADDs described in Section 5.5 as an organisational framework, we systematically analysed the *units of analysis* to identify and gather evidence of the use of any of the ADDs, along with associated decision options in the form of architectural patterns and practices. This aspect of the study was deductive, as we began with

knowledge and understanding of the ADDs from the literature relevant to the domain and the research questions we deemed a reasonable starting point. Still, we had no predefined hypotheses or assumptions about what we expected to find.

We conducted a content analysis of the data source using the method described by DeFranco and Laplante [67], which is well-suited for systematically examining software engineering data. This flexible inductive data organisation approach involves selecting suitable sources and employing coding techniques to categorise data. The method can facilitate the analysis of various software engineering artefacts and yield rich insights grounded in project-specific data, as demonstrated in our case study on the application of ADDs within a software system.

We identified patterns and themes, coded contextual information and findings to support sensemaking [89], and developed broader categories for the system's discovered architectural properties. We consulted additional literature to aid and validate our understanding of the phenomena encountered as needed. Our methodology was an iterative, continuous process of ADD identification and recovery, interpretation, categorisation, and modelling.

5.4.6 Modelling

Following the content analysis, we modelled relevant parts of the system using the same technique as in work described later in this doctoral thesis [69], [71]. We used Codeable Models [78], a Python tool for specifying metamodels and model instances. These formal architectural models consist of nodes and connectors that represent components, pipelines and process steps, along with their relationships, and can be rendered in UML using PlantUML [79]. Figure 5.2 depicts a component view of part of the modelled RL system, and Figure 5.3 a process view. Modelling the system during content analysis enhances

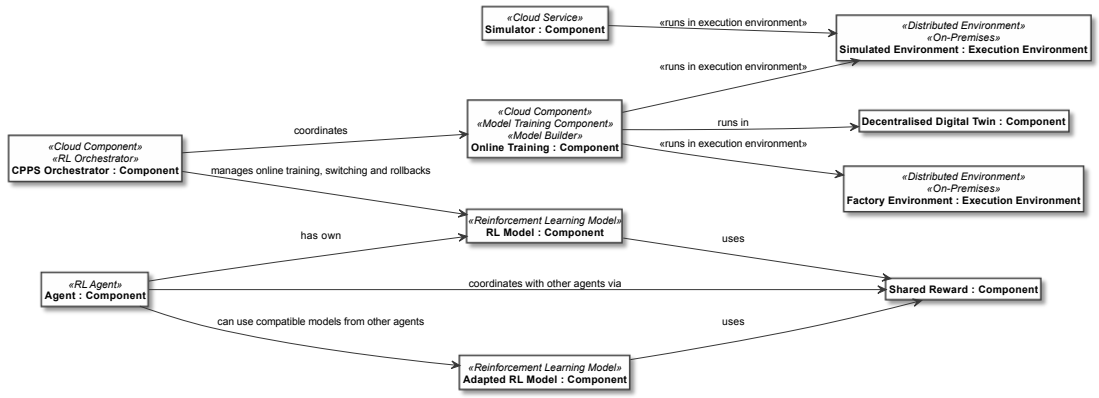

Figure 5.2: An example component model UML diagram for a *unit of analysis* of the *study object*.

the clarity and effectiveness of our case study. Simultaneously, the modelling process provides a technical reference, facilitates transparency and traceability of our process, and helps us relate our findings to the source material. A further advantage of modelling

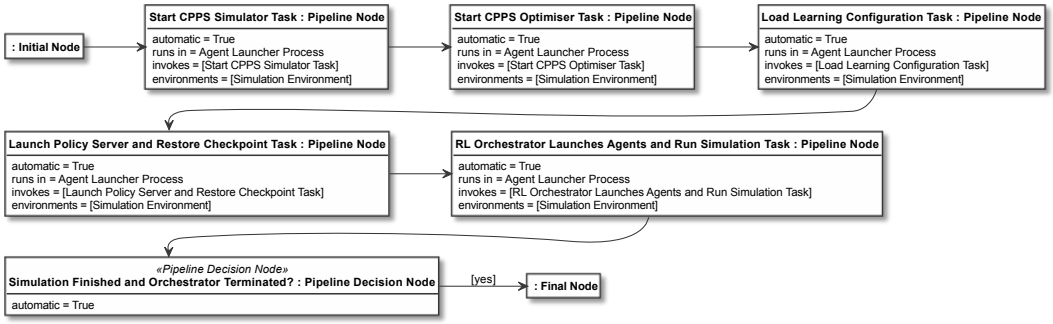

Figure 5.3: An example process model UML diagram for a *unit of analysis* of the *study object*.

during our analysis is that it can be used for validation. By formalising our understanding of the system and the use of ADDs, we provide an additional way to verify that we correctly understand the system, in combination with the clarification and disambiguation provided by industry experts during our discussions.

Owing to confidentiality agreements, the sources, that is, the *study object* itself and its *units of analysis*, cannot be shared. However, by generating UML visualisations of modelled aspects of the CPPS, we can communicate the system architecture's salient features without violating confidentiality. Please note that we did not model the entire system, but only those *units of analysis* that provided evidence for the use of ADDs and decision options. Our models were simplified to the minimal level of detail required to understand the *units of analysis*. Likewise, specific technologies were mainly omitted. Some aspects, such as component names, were renamed to protect intellectual property.

For replicability and transparency, our models, model visualisations, and an overview of our chain of evidence from the ADDs to our findings are available in our replication package [149]. We continuously validated our results through triangulation across the range of source *units of analysis* and models, cross-checking amongst the author team. Finally, we provided our results to Siemens for evaluation as an additional validation measure.

5.4.7 Ethical Considerations

Confidentiality and the handling of sensitive data were maintained at all times. There was no direct human participation in the case study beyond the usual working relationship with Siemens, who provided informed consent and had the opportunity to review, correct and give feedback on this study and its artefacts. No inducements were offered, and there were no known conflicts of interest.

5.5 Results

In this section, we report our results, describing our findings for each ADD. Table 5.1 shows the findings of our comprehensive analysis of ADDs for the *study object* and provides

insights into the application of ML and RL ADDs in this context.

Table 5.1: Summary of the Applicability and Use of ADDs in the System

ADD	Decision Option
MLOps M [24] ↗	No MLOps ↘ Apply MLOps E ↗
AutoML M [23] ↘	No AutoML ↗ Use AutoML ↘
Automatic Data Processing M [23] ↗	No data processing automation ↘ Data pipeline ✕ ETL pipeline ✕ Data processing component ↗
Data Processing Pipeline Tasks M [23] ↗	Data extraction ✕ Data transformation ✕ Data preparation ✕ Data validation ✕ Data selection ↗ Feature engineering E ↗ Data processing hyperparameter tuning ✕
Feature Storage M [23] ↗	Data store ↘ Feature store ↗
Data Ingestion M [23] ↗	Streamed data ingestion ↘ Data ingestion on request ✕ Data ingestion in batches ↗ Manual data ingestion ✕
Pipeline/Orchestrator Triggers M [23], [24] ↗	On-demand ↘ Commit ↘ Scheduled ↗ New training data E ↗ Model performance degradation E ↗ Data distribution changes ✕
Model Building M [23] ↗	In development tool E ↗ Model-building pipeline E ↗ Model-building component ↗
Model Building Pipeline Tasks M [23], [24] ↗	Model training ↗ Data splitting ✕ Data checkpoints E ↗ Model validation ↗ Model selection ↗ Train multiple model versions ↗ Model packaging ↘

M marks an ADD that is associated with *ML*.
R denotes an ADD that is *RL-specific*.
E denotes an ML decision option that has an *RL equivalent*.
↗, ↘ and ✕ represent an ADD or a decision option that was *applied*, *not applied* or *not applicable* to the study object, respectively.

Continued on next page...

ADD	Decision Option
	Model hyperparameter tuning [E] ↗
	Development tool facade ↘
	Development tool export ↘
Training Strategy [M] [23] ↗	Batched ↘
	Incremental ↗
	Batch-based/incremental hybrid ↘
Model Deployment [M] [24] ↗	Manual deployment ↘
	Pre-prepared pipelines ↘
	CI/CD pipeline automation ↗
Data Processing [M] [23] ↗	Batched ↗
	Real-time, stream-based ↘
Automated Integration/Delivery [M] [24] ↗	None ↘
	Build and deployment scripts ↘
	CI/CD pipeline ↗
	ML orchestrator ↗
Delivery Pipeline Tasks [M] [24] ↗	Packaging ↘
	Testing ↘
	Building ↘
	Deployment ↗
	Containerisation ↗
Data Versioning [M] [24] ↗	None ↘
	Data repository ↗
	Code repository ↗
Model Version Deployment [M] [24] ↗	Single model in production ↘
	N versions in production ↗
	Model version rollback ↗
Model Architecture [R] [8] ↗	Monolithic model using single-agent RL ↘
	Specialised models + multi-agent coordination via shared rewards ↗
	Specialised models + multi-agent market-based coordination ↘
	Specialised models + coordinator specialist + hierarchichal models ↘
	Specialised models + coordinator specialist ↘
Model Training [R] [8] ↗	Single-agent RL with parallel training of a single agent ↘
	Market-based multi-agent RL ↘
	Centralised training and execution multi-agent RL ↘
	Centralised training and decentralised execution multi-agent RL ↗
	Distributed multi-agent RL ↘

[M] marks an ADD that is associated with *ML*.
[R] denotes an ADD that is *RL-specific*.
[E] denotes an ML decision option that has an *RL equivalent*.
↗, ↘ and ✕ represent an ADD or a decision option that was *applied*, *not applied* or *not applicable* to the study object, respectively.

Continued on next page...

ADD	Decision Option
	Hierarchical RL ↘
Checkpoints R [8] ↗	No checkpoints ↘ Use checkpoints ↗
Transfer Learning R [8] ↗	No transfer learning ↘ Use transfer learning ↗
Distribution Strategy R [8] ↗	No distribution strategy ↘ Custom distribution framework for each RL algorithm ↘ Versatile distribution frameworks ↗ Distributed control-based distribution style ↘ Logically centralised control distribution style ↗ Hierarchically parallel task distribution style ↘
RL Hyperparameter Tuning R [8] ↗	No RL hyperparameter tuning ↘ Use RL hyperparameter tuning ↗

M denotes an ADD that is associated with *ML*.
R denotes an ADD that is *RL-specific*.
E denotes an ML decision option that has an *RL equivalent*.
↗, ↘ and ✕ represent an ADD or a decision option that was *applied*, *not applied* or *not applicable* to the *study object*, respectively.

5.5.1 MLOps

A major ADD is whether *MLOps* should be applied holistically throughout a project. MLOps touches on many other ADDs covered in this study, such as strategies for automating and streamlining data processing, rapid model training, deployment and performance monitoring. The *units of analysis* clearly show that the decision to adopt an MLOps approach was made. However, the findings below indicate that for many related ADDs, MLOps as a practice needs to be adapted to accommodate the distinct demands of RL, resulting in a modified form of MLOps for RL, termed RLOps.

5.5.2 AutoML

AutoML automates several ML practices that would otherwise be performed manually, such as data preparation, feature engineering, model training, hyperparameter tuning, model validation and model selection. It uses search algorithms to find optimal solutions for the various phases of the ML pipeline. In the *study object*, AutoML was not applied.

5.5.3 Automatic Data Processing

Deciding how to process the data used for model building automatically is central. Neither **data pipelines** nor **ETL pipelines** are suited to RL, since RL models are trained on environmental data and rewards rather than on training datasets. In the *study object*, a **data processing component** in the form of an environment interpreter that provides

environment data to agents is used, supporting *Automatic Data Processing*. An alternative implementation considered for a future version of the system is the **replay buffer** [2]. Replay buffers are widely used to retain and replay historical data to prevent loss. This alternative would process production data and provide it to the training process.

5.5.4 Data Processing Pipeline Tasks

Deciding on *Data Processing Pipeline Tasks* is a follow-on ADD from *Automatic Data Processing* if, for instance, a **data pipeline** or **data processing component** was used. In RL, tasks related to ML-specific training data, such as **extraction**, **transformation**, **preparation**, **validation**, and **data processing hyperparameter tuning**, are not applicable. During manual experimentation, **data selection** of environment features is used. This data is used for offline RL [150] and an RL-adapted form of **feature engineering**, which focuses on creating an effective state representation for imitation learning [2]. Offline RL occurs when an agent learns from past data without interacting with the environment, and imitation learning develops a policy from expert demonstrations, potentially surpassing the expert's performance.

5.5.5 Feature Storage

The *Data Processing Pipeline Tasks* decision option of **feature engineering**, as applied during manual experiments, prompts the architect to consider *Feature Storage* options. These include a simple **data store** or a specialised **feature store**, the latter of which was applied in the *study object*. In the context of RL, a feature is a specific attribute or property of the environment that the agent can use to make decisions. The **feature store** option facilitates persistence and access to feature data and, in the *study object*, the data flows back from the environment interpreter to the **feature store** via a trigger (see the *Pipeline/Orchestrator Triggers* ADD in Section 5.5.7) to be used for offline RL and imitation learning.

5.5.6 Data Ingestion

Another key ADD is *Data Ingestion* to enable *Data Processing* and consequently *Model Training*. In this RL-based CPPS case, **data ingestion on request** and **manual data ingestion** are irrelevant since it only makes sense to ingest data when there has been a change in model performance or the environment (such as the factory layout) – see the *Pipeline/Orchestrator Triggers* ADD in Section 5.5.7. Thus, it only remains to be decided whether data is **streamed** or **batched**. Both are feasible, but the **batched** option has been applied in our case study. Data resulting from exploring the action space is provided via a trigger, saved in a **feature store** for offline RL and imitation learning, and is then fed into an automated pipeline in batches for model training.

5.5.7 Pipeline/Orchestrator Triggers

ML pipelines, orchestrators and model-building components can be started via *Pipeline/Orchestrator Triggers*. Triggers can be **on-demand** (not used in this project) or based on external events, such as a **commit** to the code base (not used either). In RL, triggering on **changes to the data distribution** does not make sense since the system does not deal with labelled datasets with a given distribution, as in supervised learning. In our case study, a modified version of the **new training data** trigger was applied based on action space (e.g. after agent actions or when robot characteristics are otherwise updated) or environment changes (e.g. when the factory layout changes). **Model performance degradation** also triggers pipelines in the *study object* and is achieved via agent performance monitoring (necessitating an RL-specific adaptation), triggering a training pipeline or component or when a better model becomes available, triggering a deployment pipeline or model serving component. A model serving component is also triggered according to a **schedule**.

5.5.8 Model Building

A core ADD in an ML project is *Model Building*. Options include using **development tools**, such as **computational notebooks**. Alternatives include using a **model-building pipeline** or a **model-builder component**. The *study object* uses all three: a **development tool** for running manual experiments, a **model-building pipeline**, and manual experimentation uses a **model-building component** for model training.

5.5.9 Model Building Pipeline Tasks

If a **model-building pipeline** is chosen for the *Model Building* ADD, as in our case study, then the *Model Building Pipeline Tasks* ADD is for deciding which tasks are performed in that pipeline. The model-building pipeline and other components perform **model validation**, **model selection** from an RL model registry, **training of multiple model versions** and an RL version of **hyperparameter tuning** (see the *RL Hyperparameter Tuning* ADD in Section 5.5.22). **Model-building components** perform **model training** and use **data checkpoints** (although modified for use in offline RL). **Data splitting** (e.g. into training and test data) does not apply to this project since it does not use datasets for supervised/unsupervised learning. Model **packaging** is not applied here because models are loaded directly from an RL model repository and into an agent via an orchestrator component. Finally, neither a **development tool facade** nor **export** is used since development tools are not integrated into the automated flow.

5.5.10 Training Strategy

Model training, discussed with the *Model Building Pipeline Tasks* ADD in Section 5.5.9, entails a follow-on decision on which *Training Strategy* to adopt. The training strategy may be **batched**, **incremental** or a **hybrid** of the two. In our case study, models are built

incrementally in response to automatic triggers described for the *Pipeline/Orchestrator Triggers* ADD in Section 5.5.7.

5.5.11 Model Deployment

Once a model is trained, it needs to be deployed for use. Our case study uses neither **manual deployment** nor semi-automated **pre-prepared pipelines** – though both are possible, neither practice is ideal for an Industry 4.0 CPPS. Instead, **CI/CD pipeline automation** is used to automate a model serving component.

5.5.12 Data Processing

Data Processing is paramount in ML. It involves taking data that was already ingested and providing it to, e.g. a **model-building component** for use in model training. Processing can occur in **batches** when required for training, as in the *study object* where data is provided in batches with triggers and read in batches for offline and imitation learning or in **real-time as a stream**, which is not implemented in our case study.

5.5.13 Automated Integration/Delivery

Automated Integration/Delivery involves deciding on the level of automation for integration and delivery. The most straightforward choice, **no integration or delivery automation**, is not applied, nor is manually performing deployments **using build and deployment scripts**. Instead, the system makes use of a **CI/CD pipeline** (for building and deployment) and an **ML orchestrator**. The orchestrator loads models from a model repository into agents. It executes the agent and evaluates its actions, observations and rewards in a simulator.

5.5.14 Delivery Pipeline Tasks

If **CI/CD pipelines** are used, as they are in the *study object*, then *Delivery Pipeline Tasks* are a consideration. Typical tasks include **packaging** (e.g. of models), **testing** delivery tasks (e.g. for A/B tests or canary tests), **building** (e.g. of executables), the **deployment** of artefacts into a target environment and **containerisation** (e.g. of models or components). In our case study, model evaluation is part of the training pipeline, not the delivery pipeline. Model building is part of the training pipeline and not the delivery pipeline. Our case study uses **deployment** tasks as specified in a deployment configuration to deploy a simulator and a **model-building component**. It also performs **containerisation** of Docker [33] images and stores them in a container registry, rather than packaging models.

5.5.15 Data Versioning

Part of data management involves *Data Versioning*. ML or RL-relevant data can be stored and versioned in a dedicated **data repository**. Likewise, ML or RL code can

be stored in a **code repository** like GitLab. In the *study object*, both practices were applied with source code stored in a **code repository**, environment data stored in the **feature store** described in Section 5.5.5, model metadata stored in an ML metadata store, which is a kind of **data repository**, and models stored in a model registry.

5.5.16 Model Version Deployment

Fundamental to more detailed ADDs (e.g. *Model Architecture* for RL in Section 5.5.17) is deciding on *Model Version Deployment*. We note that a **single model in production** was not applied since the system uses multi-agent RL. On the other hand, the system was designed to support **n versions in production** – to support staged releases and upgrade subsets of cyber-physical production units (CPPUs) – and also **model version rollback** via a model registry.

5.5.17 Model Architecture

In RL, the correct choice of *Model Architecture* is more complex than in ML. One can either make use of a **single monolithic model** for centralised decision-making or multiple **specialised models** with various forms of coordination. Coordination may be achieved with **shared rewards**, be **market-based** or can be achieved with a **coordinator specialist**. If **specialised models** and a **coordinator specialist** are selected, then **hierarchical models** may also be used. Our analysis shows that the *study object* uses **specialised models with multi-agent coordination via shared rewards**.

5.5.18 Model Training

For the *Model Training* ADD, decision options include patterns and practices incorporating a range of combinable strategies, including **single-agent RL**, **multi-agent RL**, **hierarchical RL**, **market-based RL**, **parallel training** and **centralised/decentralised training and execution**. In our study, the system uses **centralised training and decentralised execution multi-agent RL**: the models are centrally trained in a model-building component (see the *Model Building* ADD discussed in Section 5.5.8), and multiple decentralised digital twin agents are executed.

5.5.19 Checkpoints

Checkpoints are a binary choice and were applied in the *study object* to facilitate resumable training. Resumable training requires saving training parameters, state, and data at given intervals during training, allowing resumption later. This practice can enhance training robustness by enabling the training process to recover with minimal resource waste in the event of interruption, and it also facilitates model versioning during training.

5.5.20 Transfer Learning

Transfer Learning is a binary ADD representing a technique for using knowledge gained from solving one problem to solve a related but different task. In the *study object*, it is used by fetching pre-trained models from the model registry and retraining them to reduce overall training time.

5.5.21 Distribution Strategy

A suitable RL *Distribution Strategy* can improve training efficiency and scalability, and reduce training time. Depending on specific needs, various combinations of factors characterise the decision options, including frameworks, control methods and levels of parallelisation. The *study object* uses **versatile distribution frameworks** with **logically centralised control distribution**.

5.5.22 RL Hyperparameter Tuning

In RL, similarly to ML, the practitioner can apply *RL Hyperparameter Tuning*, which is considered distinct from the **data processing hyperparameter tuning** decision option described in Section 5.5.4 since despite the common overall goal of optimising model performance, there are some key differences, such as the optimisation of a reward function or policy over time across multiple episodes rather than a performance metric like accuracy, with distinct training runs producing a completely new model. Our case study uses **RL hyperparameter tuning** during base training, i.e. in the agent's initial training phase.

5.6 Discussion

In this section, we assess, interpret and discuss the results of our rigorous qualitative case study regarding the research questions defined in Section 5.1.

RQ$_{2.4}$: *To what extent are known ML and RL ADDs applicable to RL in a CPPS context?* After analysing the system, modelling 15 *units of analysis*, and considering 22 ADDs with 86 decision options from the literature, we identified evidence for 21 ADDs and 37 design decisions in a concrete, RL-based Industry 4.0 CPPS. Only one ML ADD was not applied (but could have been). The most important observation is that all ML ADDs were applicable, and all but one were indeed applied, indicating a high level of cross-compatibility between ML and RL practices. All six RL-specific ADDs were applied to the *study object*, in line with our understanding of their use.

In practice, the only ADD out of the 16 top-level ML ADDs that was not applied was *AutoML*, although it is not unsuited to RL and still could have been used in specific circumstances [151], [152]. Practitioners should evaluate whether AutoML could offer benefits in their specific use cases, particularly for tasks that do not require highly specialised models. In our case study, it was not applied, but that does not preclude its applicability in other systems.

More generally, understanding the reasons for choosing specific options over alternatives is outside the scope of this study. We only considered project artefacts and known and evidenced ADDs, but this does not rule out other decision options or as-yet-unknown ones that may be implemented over the system's lifetime.

RQ$_{2.5}$: *Which ADD decision options typically used in ML and RL apply directly to a real-world CPPS that utilises RL?* We considered a total of 86 decision options, 63 of which were ML-related and 23 RL-specific. Out of 63 ML decision options, 22 were directly applied, 22 ML decision options were not applied (but could have been), and 11 ML decision options were not applicable. See **RQ$_{2.6}$** below for a discussion on the remaining eight ML decision options, which were applicable with modification. These results suggest that many existing MLOps practices can be transferred to RL-based industrial systems without modification. Out of 23 RL-specific decision options, seven were applied, 16 were not applied (but could have been), and all were applicable. These results indicate that the *study object* closely conformed to our prior knowledge of RL decision options and was highly compatible with, and relevant to, known practices.

Noteworthy is the varying emphasis placed on specific decision options, evident in Table 5.1, depending on whether the system is geared more towards ML or RL. For instance, ML decision options that are not suited to RL tend to be data-centric, such as with *Automatic Data Processing* practices (e.g. use of a **data** or **ETL pipeline**), or *Data Processing Pipeline Tasks* (e.g. **data extraction**, **transformation**, **preparation** and **validation**). RL-specific decision options tend to focus on *Model Architecture* (**agents** and their specific roles), *Model Training* (**centralised**/**decentralised training** and **execution**) and model *Distribution Strategy* (**centralised** or **distributed**). These differences are evidenced in the *units of analysis* and are thus relevant when planning resource allocation (e.g. developers' time and effort).

RQ$_{2.6}$ *Is it necessary to adapt ADD decision options typically used in ML for use with RL in a CPPS context?* Out of 63 ML decision options, eight ML decision options were applied in the form of an RL-equivalent, that is, a version of the original option modified to suit RL-specific needs. An interesting observation is that some of these decision options include practices that at first glance may not appear to have an RL-analogue, such as the *Data Processing Pipeline Task* **feature engineering**, and *Pipeline/Orchestrator Triggers* on **new training data**.

Given that all top-level ADDs from the literature were applicable in some way, the main differences are noticeable at the level of the decision options. Categories emerge at this level of granularity regarding their RL compatibility. We derived new categories to aid understanding of the use of ML, and RL ADDs in CPPSs: **inapplicable ML decision options**, **unused ML decision options**, **ML decision options with RL equivalents** and **directly applied ML decision options**. The rationale for our classification can be independently verified via Table 5.1 and our replication package [149].

(1) **Inapplicable ML Decision Options** are marked with ✕ in Table 5.1, for example **data pipeline**, **data distribution changes** and **data splitting**. In total, we identified 11 such decision options. These decision options can be disregarded early in the architectural design process, as they are not compatible with RL and do not provide an RL-adapted

equivalent. Recognising inapplicable options helps practitioners focus their efforts on relevant solutions, but it is crucial to regularly reassess these options as technology and requirements evolve.

(2) **Unused ML Decision Options**, denoted by ↘ for ADDs marked M in Table 5.1, include **use AutoML**, **data store** and **model packaging**. Twenty-two such decision options were found. These decision options are compatible with RL without requiring RL-specific modifications, but were not utilised in the *study object*. They represent potential opportunities for future enhancements or system extensions. Practitioners should remain aware of these options and regularly reassess their applicability as systems mature.

(3) **ML Decision Options with RL Equivalents** are marked with E in Table 5.1, for instance, **new training data**, **model-building pipeline** and **data checkpoints**. Eight of these decision options were identified. These decision options may appear, at first glance, to be incompatible with RL and, therefore, may be prematurely disregarded. However, practitioners should note that they can be modified to suit RL better. The fact that all eight ML decision options were modified and used for RL highlights the need for flexibility. Practitioners should be prepared to adapt standard ML practices to fit the RL paradigm. This category is particularly valuable for practitioners transitioning from traditional ML to RL in industrial settings. It provides a bridge between familiar ML practices and RL-specific requirements.

(4) **Directly Applied ML Decision Options** are indicated by ↗ (without E) for ADDs marked M in Table 5.1, and include **data processing component**, **data selection** and **feature store**. We noted 22 such decision options. Practitioners should be aware that the ML-related decision options in this category are directly compatible with RLOps. They should be encouraged to use them where appropriate.

It is also worth noting a category of decision options that we already knew to exist - those belonging to RL-specific ADDs. The application of all six RL-specific ADDs indicates the importance of considering RL's unique characteristics in industrial settings. The associated decision options also offer the most straightforward path to implementation. Practitioners should consider prioritising them to achieve rapid benefits and establish a foundation for more complex integration later in the system's lifecycle.

Our results demonstrate that RL practitioners wishing to implement RLOps have a rich set of ADDs at their disposal. This breadth of choice enables tailored solutions that address the unique challenges of RL in Industry 4.0 CPPS environments. By carefully considering the applicability of various ADDs and decision options and by addressing RL's unique challenges in industrial settings, practitioners can leverage these technologies to enhance efficiency, adaptability and decision-making in their operations. Practitioners are advised to maintain flexibility and continually reassess options, as a commitment to ongoing adaptation is essential in this rapidly evolving field.

5.7 Threats to Validity

In this section, we consider threats to validity according to Wohlin et al. [85] and Yin [89].

5.7.1 Internal Validity

Our selection of the *study object* and its context (for instance, its domain), as well as its level of MLOps/RLOps maturity, could have invalidated or skewed our results. However, we consider the *study object* representative of the broader industry, and this assertion is supported by the wide range of ML and RL practices documented in the case study.

If a software project is still in development at the time of a study, changes over time within the *study object*, e.g. in team composition, system architecture or technologies, could influence findings. We do not consider this threat significant, as the project was in an advanced stage of development and any relevant architectural decisions had already been made. In any case, this factor applies to all projects, as all development and production systems evolve until they are decommissioned. Our case study provides an accurate temporal snapshot, and it is worth noting that none of the documented practices or ADDs contradicts or is incompatible with any other.

5.7.2 External Validity

The generalisability of our findings is a potential threat to external validity if the *study object* is not representative of the broader industry or the practical use of MLOps/RLOps. In this case, the findings may not apply to other contexts. Despite the novel nature of the system, we were provided with many data sources for triangulation, and the *study object* represents the state of the art. We consider the *study object* a typical case due to the broad range of evidenced practices discovered. We expect our results to apply to other cases with similar characteristics and anticipate that our findings will be relevant to those cases as well. Still, it would be interesting to conduct other case studies in similar and differing contexts for comparison (see Section 5.8).

5.7.3 Construct Validity

We set out to understand how ML ADDs are applied in practice and how they relate to RLOps. Our definitions and understanding of MLOps and RLOps ADDs might not align with their use in the *study object*, leading to inconsistent interpretations and less convincing results. To mitigate this threat, we clearly, consistently and comprehensively defined all technical terms as we understood them, used standardised terminology (ubiquitous language [153]) and ensured the author team and partners from Siemens had a shared understanding. We also formally modelled the relevant parts of the system and generated UML visualisations of our models for comparison, documentation and validation. Our findings indicate that our understanding of the ADDs matched their use in the *study object*, thus confirming their relevance and the validity of the study.

5.7.4 Reliability

A further risk is that the methods used in our prior work, as described in Section 5.4.1, to identify practices, ADDs and decision options, may not have accurately captured the nuances of relevant practices in the real world. The same applies to the translation of our

prior work's understanding of concepts, such as ADDs, into concrete, practical use in the case study. To mitigate this threat, we triangulated all sources of project information as a confirmatory measure and, when necessary, sought clarification from Siemens. In doing so, we successfully mapped the ADDs to phenomena encountered in the *study object*, thereby confirming the overall validity of the ADDs and the reliability of our findings.

5.8 Conclusion and Future Work

We explored the application of ML ADDs in a real-world RL CPPS. Our research questions examine the applicability of known ML and RL ADDs, the decision options directly applicable to RL, and the adaptation requirements for ML decision options in RL within a CPPS context. This study, to our knowledge, is the first of its kind and provides actionable insights into the use of MLOps and RLOps practices in Industry 4.0. It provides guidance and a reference for practitioners, linking theoretical concepts to real-world applications in RL-driven CPPSs and informing architectural design in Industry 4.0.

We categorised ADD decision options into four emergent groups: **Inapplicable ML Decision Options**, **Unused ML Decision Options**, **ML Decision Options with RL Equivalents** and **Directly Applied ML Decision Options**. This classification provides insights into applying ADDs from grey literature to real-world RLOps systems. The findings demonstrate that many ML practices are directly applicable or adaptable to RL, thereby contributing to a deeper understanding of RLOps architecture design and facilitating the implementation of effective RL practices. These insights may be of particular interest and encouragement to project teams when implementing similar Industry 4.0 CPPSs to that of the *study object*, particularly to those who are unaware that many ML practices apply to RL during architectural decision-making.

This study on ML and RL ADDs in the Industry 4.0 CPPS domain also lays the groundwork for future studies. Ideas include exploring RLOps ADDs across different industries, conducting longitudinal studies of their evolution, and performing quantitative impact analyses. Such studies could reveal sector-specific practices and new ADDs, track the maturation of RLOps practices over time, and provide empirical evidence of their effectiveness. Collectively, such studies could enhance understanding of RLOps practices in different contexts and their impact on organisational performance.

Part IV

Understanding MLOps and Reinforcement Learning-Enabled Systems

6 On the Understandability of MLOps System Architectures

MLOps is the practice of streamlining and optimising the ML workflow, from development to deployment, using DevOps (software development and IT operations) principles and ML-specific activities. Architectural descriptions of MLOps systems often consist of informal textual descriptions and graphical system diagrams that vary widely in consistency, quality, detail and content. Such descriptions often deviate from established standards or schemata, making them challenging to comprehend.

We aimed to investigate informal text and informal graphical representations of MLOps system architectures and compare them with semi-formal MLOps system architecture diagrams of those systems. We report on a controlled experiment with 63 participants investigating the understandability of MLOps system architecture descriptions using informal and semi-formal representations.

The results indicate that: (1) the understandability (quantified by task correctness and duration) of MLOps system descriptions is significantly greater using supplementary semi-formal MLOps system diagrams; (2) using semi-formal MLOps system diagrams does not significantly increase task duration (and thus hinder understanding); (3) task correctness is only significantly correlated with task duration when semi-formal MLOps system diagrams are provided.

6.1 Introduction

6.1.1 Problem Statement

MLOps is centred around rapid deployment and is similar in many respects to DevOps (software development and IT operations), but also introduces additional ML-specific activities [111]. It is a multidisciplinary engineering practice that intersects ML, software engineering and data engineering. MLOps involves specific practices for end-to-end activities, including conceptualisation, implementation, deployment, and scaling of ML products. It also utilises CI/CD, workflow orchestration, reproducibility, data processing, model and code versioning, continuous training, and monitoring, amongst other practices [4], [23], [24]. Whilst conducting previous research [23], [24], we noticed that MLOps system descriptions in grey literature are typically composed of informal textual descriptions and informal graphical system diagrams. Based on our observations, the textual descriptions vary considerably in tone, detail and content, and do not always closely correspond to the accompanying informal MLOps system diagrams. We noted that the diagrams

themselves do not usually follow consistent standards or schemata and may exhibit any of the following characteristics:

- Inconsistencies in terminology, relationships between elements and visual features.
- Lack of standardisation of terminology, relationships between elements and visual features.
- Lack of clarity.
- Varying levels of detail or abstraction.
- Omission or inclusion of relevant or irrelevant system aspects, technologies and other details.

Indeed, in practice, system architecture descriptions are customarily of an informal nature [41], [42], [43], such as ad-hoc box-and-line or box-and-arrow diagrams, and this appears to be the case still, despite the prevalence of modelling languages such as UML [66] or domain-specific languages (DSLs) for specifying architectures [154].

6.1.2 Research Objectives

We conducted an empirical study on the understandability of MLOps system descriptions. Our study aimed to determine whether and to what extent the provision of additional, semi-formal MLOps system diagrams improves understanding of MLOps system descriptions. In software architecture, informal system diagrams are graphical representations of a software system or its components, typically designed to convey high-level or conceptual information. These diagrams are often intended for quick communication and understanding amongst team members and stakeholders, as well as for initial design discussions.

Informal diagrams may lack strict adherence to standardised notation and may be hand-drawn, sketched or created with basic drawing tools. Their primary purpose is to illustrate the system's structure and major components without going into exhaustive detail. Semi-formal system diagrams, on the other hand, are more rigorous and structured graphical representations of a software system. They follow standardised notation and conventions and are often used for detailed design, documentation or analysis. Semi-formal diagrams provide a comprehensive depiction of the system, including its architecture, components, relationships and behaviours. These diagrams are typically expected to adhere to specific modelling languages, such as UML or specialised notations tailored to software architecture. They can be used for more precise communication, analysis, and as a basis for further system development and implementation.

We deemed UML a good choice for the visual system representations in this study since it is familiar, widely understood and has been utilised in the related work described in Section 6.2. UML activity diagrams for the pipeline representations were ruled out because they were deemed insufficiently precise or detailed for the study. Despite the variable nature of the formality of UML diagrams (see Rumpe and France [155], Whittle [156] and France et al. [157]), our component and pipeline diagrams still create an

improved commonality of communicating architectural decisions and intent since they are UML representations generated from coded models based on a precisely defined MLOps metamodel that ensures no variability or ambiguity in the models themselves.

Figure 6.1 depicts a redrawn excerpt of a build pipeline from an example MLOps system[1] and Figure 6.4 in Section 6.3.7 shows the corresponding semi-formal diagram for this pipeline excerpt. Please note that, like the UML-based diagram, the informal excerpt does not include any aspects unrelated to the build pipeline. In the original informal diagram, all pipelines and components are depicted in a single diagram, whereas our UML-based diagrams are separated into multiple component and pipeline diagrams. Aspects not present include the Azure Blob Storage Component, the Azure Machine Learning Pipeline endpoint, the Azure Machine Learning retraining pipeline and the Azure DevOps build pipeline.

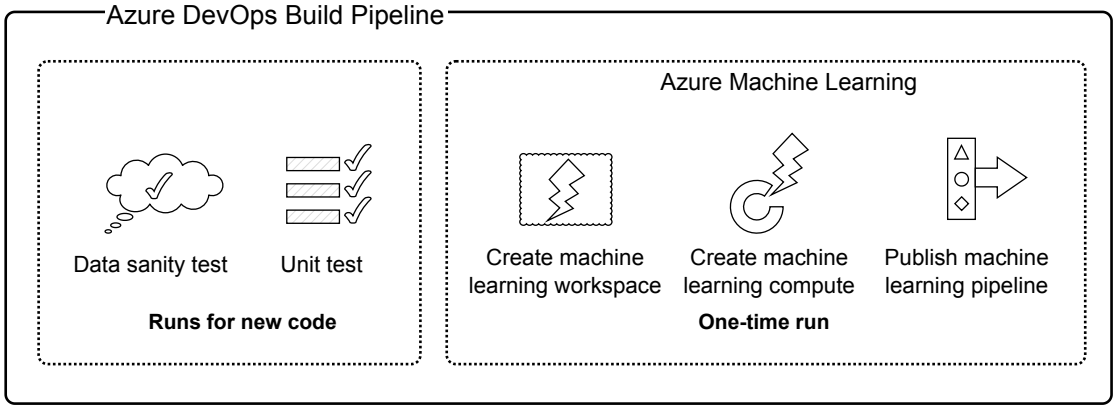

Figure 6.1: Redrawn excerpt of a build pipeline from an informal MLOps system diagram.

As in our previous work [158], [159], [160] that preceded the commencement of this doctoral thesis, we defined the experiment goal using the Goal Question Metric [161] template as follows: **analyse** MLOps system descriptions **for the purpose of** their evaluation **with respect to** their understandability **from the viewpoint of** both novice and moderately advanced software architects, designers or developers **in the context (environment) of** the Advanced Software Engineering (ASE), Distributed Systems Engineering (DSE) and Software Engineering 2 (SE2) courses offered by the Research Group Software Architecture at the University of Vienna.

We conducted an empirical study – to our knowledge, the first of its kind – as a controlled experiment on the understandability of MLOps system description representations, incorporating our novel, semi-formal, UML-based MLOps system diagrams. Our main contributions are the results of the controlled experiment: (1) that the understandability (quantified by task correctness and duration) of MLOps system descriptions is signific-

[1] Source for the complete diagram from which we took the excerpt: `https://tinyurl.com/mlops-system-ma`

antly greater using our semi-formal MLOps system diagrams; (2) that using semi-formal MLOps system diagrams does not significantly increase task duration (and thus hinder understanding); (3) that task correctness is only significantly correlated with task duration when semi-formal MLOps system diagrams are provided.

Our research makes significant contributions to the MLOps field by advocating the adoption of semi-formal diagrams as a best practice to enhance task correctness. These findings not only catalyse further scientific investigations into the cognitive aspects of correctness but also offer practical guidance to MLOps practitioners. They emphasise the potential advantages of incorporating semi-formal diagrams and highlight the significance of realistic self-assessment. Our results highlight the value of visual representations in improving correctness.

Moreover, our findings have the potential to reshape the way software architecture is taught in university courses. Leveraging semi-formal MLOps system diagrams can empower educators to enhance learning outcomes, address prevalent misconceptions, accommodate diverse learning styles and facilitate a deeper understanding of intricate architectural concepts. Furthermore, our study encourages a more comprehensive, interdisciplinary approach to teaching software architecture that aligns with the industry's evolving needs.

Overall, our work lays a solid foundation for further empirical work in the domain of architectural modelling of MLOps systems. Our results represent a positive initial step towards determining how helpful semi-formal MLOps system diagrams can be for understanding MLOps system architecture.

We have structured the remainder of this chapter as follows: Section 6.2 describes the background and motivation of this study, and compares related work to it. Section 6.3 covers our research process, system selection method, modelling method, guidelines and study design, and describes the participants, materials and experiment tasks, our dependent and independent variables, our hypotheses and the research question. Section 6.4 documents the pilot test, preparation and execution of the experiment sessions. Section 6.5 describes the processing and analysis of the dataset resulting from the experiment sessions. We also discuss the participant demographics and descriptive statistics for the experiment data before testing our hypotheses and addressing our research question based on the experiment results. Section 6.6 discusses our results, including their interpretation and the threats to the validity of our study. We conclude in Section 6.7 by discussing the impact and relevance of our research, as well as suggestions for future work that could build on our study.

6.2 Background

6.2.1 Background and Motivation

Comprehending ML systems can be more challenging than with traditional software architecture due to the intricate interplay of software development, data science and data engineering [38]. Various authors have identified significant challenges in the software structure of ML applications, such as complex software modules and their dependencies [53],

technical debt and anti-patterns [38] and design issues in ML models, such as model selection and reuse [54], [55]. These challenges highlight the intricacies of organising and decomposing ML systems, which have led to the introduction of AutoML [162], [163] and MLOps [4], [5] as a discipline related to DevOps [21] and CD [22].

It has been shown in various studies of open source systems [164], scientific literature [165] and interviews with practitioners [166] that DevOps and CD lead to a complex deployment and delivery architecture that is hard to comprehend and needs to be understood in addition to the software architecture of the system to be deployed. This problem worsens for ML systems deployed with MLOps as, in addition to the complex model and system deployments, component architectures and CI/CD pipeline behaviours, ML pipeline behaviours and ML-specific components need to be understood [4], [5], [167]. For example, consider a model retraining step in a CI/CD pipeline that must work harmoniously with model versioning and registry services. In a survey of case studies, Paleyes et al. [168] identified significant practical challenges in deploying ML systems, specifically related to ML architecture. Some relate to tools, services and architectures used in deployment; others to the potential disconnect between ML/data science experts and software engineering practices.

We investigate whether providing semi-formal, UML-based MLOps system diagrams can improve the clarity and comprehensibility of these systems, potentially bridging the gap between the two domains and thereby increasing overall MLOps efficiency. Our research motivation stems from observing inconsistencies in existing MLOps system descriptions and the scarcity of relevant studies, including controlled experiments, that focus on understanding the comprehensibility of semi-formal MLOps system models. We are unaware of any other prior empirical study that systematically investigates understanding of MLOps system architectures based on informal textual descriptions and informal graphical system diagrams, compared to semi-formal model diagrams. This study unifies these aspects and, to our knowledge, is the first of its kind. Whilst the related work described below is relevant, the studies differ from ours in several respects. By conducting this study, we aim to contribute to the development of a scientific body of work in a new research area.

Our study results confirm our suspicions and show that the understandability of informal system descriptions used in practice could be improved. Our motivation for studying whether semi-formal models can enhance the understandability of MLOps system descriptions is that MLOps systems appear more complex than ordinary DevOps-based systems and require understanding by both software engineers and relatively untrained ML/data science experts.

6.2.2 Experimental Studies

Various studies have utilised controlled experiments, but we could not find any that involved MLOps. The studies mentioned below are provided for comparison because they are similar to ours to varying degrees in describing user experiments involving software systems. However, they do not focus on MLOps, and do not compare semi-formal and informal MLOps system models and descriptions, nor do they quantify user understanding

of system properties and behaviour.

Allodi, Cremonini et al. [169] compare the accuracy with which security professionals and students with further technical education assess the severity of software vulnerabilities based on various attributes, in a study involving 73 participants. Their focus was not on comparing different system description methods but on participants' background knowledge and education. One major difference in methodology in their study is that participants were divided into three groups: students with a BSc in information security enrolled in an MSc in information security, students enrolled in an MSc in computer science and security practitioners. Another difference is that they specifically recruited students with no professional expertise.

Allodi, Biagioni et al. [170] also conducted a controlled experiment with 29 MSc students to determine how difficult it is for participants to assess system vulnerabilities when security requirements change. Thus, their comparisons were based on variations in system requirements rather than modelling variations. Unlike our study, they employed a within-subject design; however, they formulated hypotheses that were subsequently tested using statistical methods, as we do.

Labunets et al. [171] report on a controlled experiment with 29 MSc students to compare participants' perceptions of visual versus textual methods for security risk assessment in the context of effectiveness. This experiment is relevant to our study since it compares textual and visual representations. Still, it does not focus on the differences between semi-formal and informal representations, nor on system understanding. Additionally, it focuses on security aspects rather than MLOps. Regarding methodology, the authors employed a within-subject design, whereas our study uses a between-subject design. They employed Grounded Theory during the qualitative analysis phase of their process to explain potential differences between the two methods under consideration. This approach is interesting because we also used Grounded Theory in our previous work upon which this study was partially based (see [23], [24] and our research process in Section 6.3). They translated their research questions into hypotheses that were statistically tested, a methodological approach similar to ours.

A more closely related study is described by Sharafi et al. [172], who conducted an experiment with 28 participants and examined the impact of structured textual versus graphical representations on efficiency during requirement comprehension tasks. This study has some similarities with ours. It featured graphical system representations and measured task correctness and duration. Still, it differed in emphasising visual effort, the impact of native language, education and gender. It also utilised eye-tracking and focused on requirements rather than MLOps system architectures. A difference in methodology was the use of a within-subject design; however, similarities included the definition of hypotheses and the statistical testing of these hypotheses using R [173].

A similar study to ours involving a controlled experiment is that of Heijstek et al. [174]. They conducted a controlled experiment with 47 participants from industry and academia to study whether visual or textual artefacts are more effective at communicating ADDs to software developers. They used UML diagrams and informal textual descriptions and found that: (1) neither diagrams nor textual descriptions proved to be significantly

more efficient in terms of communicating ADDs; (2) diagrams are not more suited to convey ADDs of a topological nature; (3) participants who predominantly used text scored significantly better than their counterparts. They also found that diagrams were not able to alleviate the difficulties non-native English-speaking participants had in extracting information from the documentation. Some differences between their study and ours are that all participants assessed both representations (i.e. a within-subject approach was followed), the focus was not on MLOps and data was collected by filming participants and encouraging them to think aloud. A similarity to our approach included the definition of hypotheses for statistical testing and the use of questionnaires.

Another similar study is the controlled experiment involving student participants by Jolak et al. [175]. Similarly, they compared textual and graphical (UML) representations. They focused not just on understanding but also on explainability, recall and communication. Their work was in the context of a mobile application, whereas our study focuses on MLOps. Like our study, they followed a between-subject design to minimise learning and transfer effects, gathered descriptive statistics to understand the data, and statistically tested their hypothesis. Their results favoured the graphical over the textual representations.

6.2.3 Studies on System Properties and Behaviour

There is extensive related work focusing on quantifying system properties and behaviour or on providing alternatives or enhancements to UML, which bear similarities to our study but again lack a focus on MLOps. Some notable examples are described below. These studies focus on quantifying architectural properties rather than on users' understanding of system descriptions, even when the properties are related to understanding. In contrast, we are interested in comparing semi-formal and informal descriptions of MLOps systems.

The selected studies described below propose various methods to model system architectures, provide alternatives or enhancements to UML or derive improved architectures. Although some of the artefacts resulting from these contributions may be generally applicable, we are explicitly focused on MLOps system architecture. Nevertheless, these studies remain relevant to our work, as our study involves developing a Python-based metamodel for MLOps systems and generating UML-based visualisations.

Pradella et al. [176] describe a new temporal logic language that combines UML notation with a formal semantics, providing a formal notation that allows developers to model non-critical system aspects in either UML or using formal notation. Modelling system aspects is relevant to our work, as is the provision of complementary formal or semi-formal notation, since we also used our own existing DevOps metamodels for modelling systems in UML in this study and extended them with MLOps-specific metamodel types; however, the focus of their work was not within the context of understandability and MLOps.

Lavazza et al. [177] describe an approach in which developers model systems in UML and the models are then automatically translated into a formal notation that can subsequently be used to verify various system properties, such as safety, utility and liveness. They propose enhancements to UML and provide a formal semantics for some UML constructs. Again, the focus on system properties and formal or semi-formal modelling is relevant,

but this study did not specifically focus on understandability or MLOps systems.

Rodano and Giammarco [178] use a general logical notation to formalise architectural models of systems, and quantify system characteristics and quality attributes (such as performance) to ascertain the quality of architectural models (as opposed to quantifying user understanding). They suggest that their general notation can be combined with many system architecture tools and adapted to suit specific architectural frameworks or projects rather than MLOps systems directly.

Li and Horgan [179] discuss architectural designs and quantify architectures for software systems. They present a method for constructing formal models of software architectures and simulating them to predict behaviour, reliability and performance. They then use the quantified simulation results to evaluate alternative software architectural designs. This method is interesting since we are also concerned with system properties and behaviour. However, we would like to determine users' understanding of systems using informal text and informal diagrams, together with semi-formal architecture-view descriptions, rather than generating alternative architectures or predicting system qualities.

6.3 Experiment Planning

6.3.1 Research Process

Figure 6.2 provides an overview of our overall research process. In the pursuit of our research objectives, the research methodology unfolded through a series of carefully orchestrated steps. These steps encompassed diverse aspects, from initial data collection, as described in Sections 6.4 and 6.5.1, to the rigorous analysis and interpretation of the experiment's outcomes, documented in Sections 6.5 and 6.6.

Our journey began with us systematically and iteratively scouring online resources for MLOps systems. Through each iteration, we applied meticulous inclusion-exclusion criteria (described in Section 6.3.2) until three suitable MLOps systems emerged, forming the foundational pillars of our study.

Subsequently, we outlined the guidelines for our research and the experimental design, which we describe in Sections 6.3.4 and 6.3.5, respectively. This crucial stage laid the groundwork for the controlled and structured execution of the study, ensuring the integrity of the results.

A pivotal phase of our research involved the intricate modelling of the selected MLOps systems, as described in Section 6.3.3. This modelling endeavour was intertwined with developing a comprehensive metamodel for MLOps, and a set of intricate MLOps system models comprising component and pipeline diagrams and providing a detailed and holistic representation of MLOps systems.

To provide study participants with the necessary resources for successful engagement with the experiment, we devoted much effort to preparing and producing the study materials, which we document in Section 6.3.7. We meticulously crafted this material, including the information sheets and experiment task sheets, to facilitate a comprehensive understanding of MLOps systems.

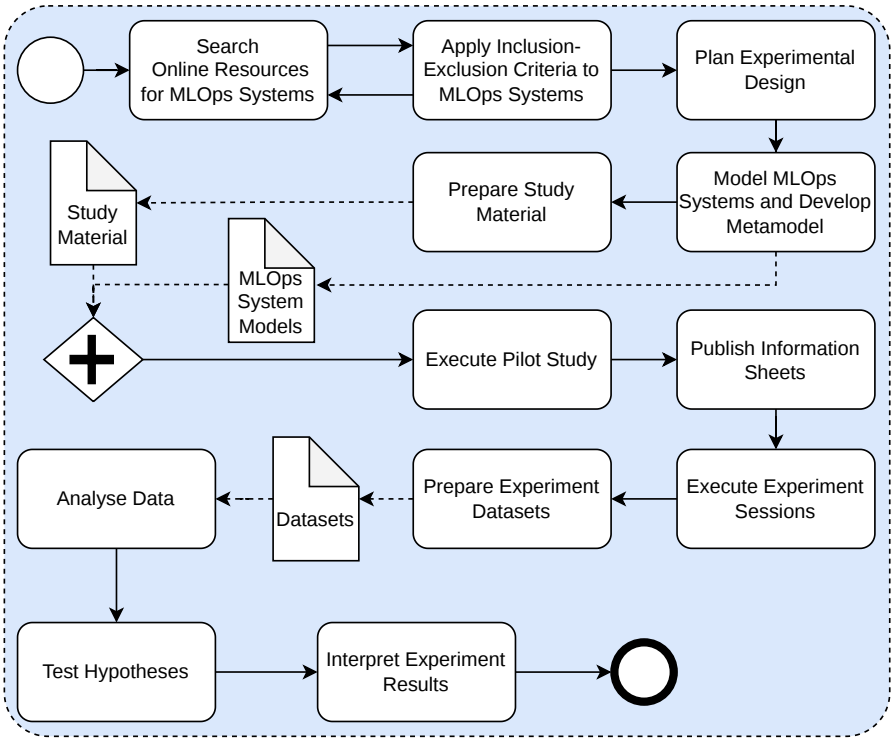

Figure 6.2: Overview of the research process followed in this study.

A critical milestone in our research journey was the execution of a pilot study, which we describe in Section 6.4.1. This pilot study was a vital validation step, using the study material and MLOps system models to assess their clarity, comprehensibility and suitability for the main experiment.

To maximise accessibility and ensure participants were well-prepared, we published the experiment information sheets on our online learning platform (see Section 6.4.2). This platform served as a central hub for disseminating essential information to participants.

The core of our research was embodied in the execution of the experiment sessions. We conducted the experiment sessions meticulously and precisely, adhering to predefined procedures and protocols, as described in Section 6.4.3.

After the experiment sessions, the study entered the data compilation and preparation phase. We carefully curated the dataset for analysis. We document these activities in Section 6.5.1.

With datasets in hand, we performed comprehensive data analysis, unearthing valuable insights, demographic information, quantitative findings and meaningful statistical relationships. This analysis, which we detail in Section 6.5, provided the groundwork for enabling the rigorous assessment of observed differences and the testing of our research hypotheses.

In the final stretch of our research journey, we diligently interpreted the experiment results. This interpretive phase allowed us to synthesise the accumulated knowledge and

insights, gaining a comprehensive understanding of the research's overarching outcomes and implications. These concluding insights provided the foundation for our contributions to the field of MLOps and the broader impact of our research. The interpretation and implications are discussed in Sections 6.6.1 and 6.7.1, respectively.

6.3.2 System Selection Method

We utilised popular search engines such as Google and DuckDuckGo, along with topic portals such as InfoQ and DZone, to source relevant MLOps systems. The selection of each system adhered to predefined inclusion and exclusion criteria.

Inclusion criteria encompassed:

- Commercial products/platforms, open source systems, workflows featuring architectural descriptions, and comprehensive examples or tutorials that facilitated the derivation of system architectures.
- Demonstrations or example projects by individuals or companies with high ratings on GitHub.
- The need to originate from reputable authors, companies or publishers.
- The need for practitioners to be the intended audience.

Conversely, exclusion criteria encompassed:

- Student projects, including those from universities, coursework, pet projects or experimental examples.
- Open-source projects categorised as tools or libraries, distinct from MLOps implementations or system descriptions.
- Tutorial articles lacking examples or containing only code snippets.
- Sources focusing solely on specific, detailed aspects of the ML workflow, as opposed to the overall workflow or system.
- Sources describing the ML workflow exclusively in a theoretical context (e.g. for informational or educational purposes).
- Sources that served as advertisements or did not describe real architectural examples applicable in an industrial context.
- Sources detailing the consulting approaches of companies without providing accompanying example projects.

The first selected MLOps system[2] was a guide based on Amazon Web Services (AWS) intended for data scientists and ML engineers looking to implement DevOps principles

[2] https://tinyurl.com/mlops-system-ap

in the context of ML systems, i.e. MLOps, facilitating the automation and continuous monitoring of every phase in the construction of ML systems and covering areas such as integration, testing, release management, deployment and infrastructure administration.

Our second selected MLOps system[3] by Google describes its level 1 architecture, which aims to ensure uninterrupted model training by automating the ML pipeline and facilitating the seamless deployment of model prediction services. The pipeline includes automated data and model validation procedures to streamline the incorporation of new data for model retraining within a production environment. It encompasses triggers and metadata management to ensure efficient workflow automation.

Our final selected MLOps system[4] was a reference architecture outlining the procedures for establishing a CI/CD and retraining pipeline for an AI application using Microsoft Azure DevOps and Azure Machine Learning. A practical demonstration of this architectural framework was accessible via a reference implementation on GitHub.

The selected MLOps systems are representative of real-world MLOps systems. We achieved this through the meticulous application of our inclusion-exclusion criteria, which guided the selection of systems offered by notable vendors, including AWS, Google and Microsoft.

6.3.3 Modelling Method

To create our models of the selected systems, we utilised Codeable Models [78], an effective Python modelling tool that allows us to specify metamodels and model instances. In this study, we initiated the development of an MLOps system architecture metamodel for specifying components, connectors and their relations. Our metamodel serves as the cornerstone for the semi-formal representation of MLOps systems, utilising modelling primitives derived from our understanding of various aspects of the MLOps systems described in Section 6.3.2.

The semi-formal, UML-based MLOps system representations in this study typically encompass component and pipeline views, each consisting of nodes and connectors of various types. In component views, nodes represent different component types, with connectors delineating their relations. These component nodes may represent various entities, including execution environments, cloud components, platforms, orchestrators, repositories, clusters or pipelines. Connectors within a component view articulate relationships, such as artefact providers, data triggers, deployment targets and pipeline triggers.

Pipeline views describe the internal workings of a pipeline and consist of an initial node, an intermediate sequence of nodes and a concluding node. The relations between these nodes establish the execution order. To address decisions or branching within pipelines, we utilise fork nodes followed by parallel pipeline nodes that culminate in a joint node. This approach represents diverse pipeline aspects and steps, encompassing pipeline triggers, data processing, model training, container image creation, testing, model registration and deployment. Pipeline nodes also house metadata, such as automatically

[3] https://tinyurl.com/mlops-system-gm

[4] https://tinyurl.com/mlops-system-ma

invoked pipeline tasks and execution environments.

To visualise our models, we employed PlantUML [79] to generate the corresponding UML diagrams. Examples of component and pipeline diagrams for the Microsoft Azure DevOps and Azure Machine Learning system are described in detail in Section 6.3.7 and may be found in Figures 6.3 and 6.4, respectively.

6.3.4 Guidelines

When planning our study, we followed the template provided by Jedlitschka et al. [83], which details robust guidelines for empirical research in software engineering. Additionally, we followed Kitchenham, Pfleeger et al. [84], who offer overarching guidelines for conducting software engineering experiments and provide recommendations related to the design, implementation, analysis and presentation of empirical research studies. We also follow Wohlin et al. [85], who go into more detail and discuss statistical tests and their suitability for various study types and Juristo and Moreno [86], who also offer valuable insights into conducting empirical research in software engineering. We used the robust statistical method guidelines for empirical software engineering by Kitchenham, Madeyski et al. [87] as a basis for evaluating the data from the experiment sessions.

6.3.5 Study Design

Conducting a controlled experiment to evaluate the impact of providing semi-formal MLOps system diagrams has distinct advantages, as it ensures causal inference and facilitates replication. This method allows for the collection of quantitative data and the generalisation of findings to broader populations. Other techniques, such as observational studies, eye tracking, surveys, comparative studies, qualitative interviews and usability studies were considered but ruled out due to limitations in establishing causality and controlling biases. Given the need for reliable, objective and causal inferences, a controlled experiment was deemed the most suitable approach for this study, as it provides necessary control over variables and ensures the validity of the results.

We deliberately chose a between-subject [82] experimental design, which involves distinct groups of participants exposed to different conditions, for several reasons. Firstly, within-subject designs, in which the same participants experience all conditions in sequence, are susceptible to order effects such as practice effects and fatigue decline, which can confound results. Using a between-subject design, we ensured that each group encountered only one condition, effectively mitigating these order-related biases.

Secondly, within-subject designs can be problematic in studies where participants learn from the first condition and apply that learning to subsequent conditions. Opting for a between-subject design allowed us to start each group with a clean slate, minimising the influence of habituation on our findings.

Thirdly, random group assignment enhanced internal validity, mitigating bias due to participant demographics, such as prior knowledge or exposure to different experiment material. Finally, equal incentives for all participants reduced the likelihood of participants biasing the survey and self-assessment responses.

6.3.6 Participants

The study consisted of controlled experiment sessions with 72 participants. After applying the objective inclusion-exclusion criteria described in Section 6.5.1, we included 63 unique participant contributions for evaluation. Notably, much widely-cited related work either used even smaller sample sizes (see Allodi, Biagioni et al. [170]: $n = 29$, Labunets et al. [171]: $n = 29$, Sharafi et al. [172]: $n = 28$ and Heijstek et al. [174]: $n = 47$) or similar sample sizes (see Allodi, Cremonini et al. [169]: $n = 73$).

We randomly assigned each participant to one of two groups: a control group ($n_{\text{CONTROL}} = 31$) and an experimental group ($n_{\text{EXPERIMENTAL}} = 32$), hereafter referred to as CONTROL and EXPERIMENTAL, respectively. Each group had a different experiment sheet, as detailed in Section 6.3.7. All participants were either BSc or MSc students enrolled in at least one of three different courses spanning two semesters:

- In the summer and winter semesters of 2022, we invited students enrolled in Distributed Systems Engineering (DSE)[5][6] to participate. This course is optional at the bachelor's and master's levels.
- In the summer semester of 2022, students enrolled in Advanced Software Engineering (ASE)[7], which is a mandatory master's-level course, could also take part.
- In the winter semester of 2022, students enrolled in Software Engineering 2 (SE2)[8] were also able to participate. SE2 is a mandatory bachelor's-level course.

We obtained written consent from all participants, participation was entirely voluntary, and we awarded students extra credit in the form of bonus points on top of the standard course points as an incentive to participate. The material covered in the study was related to the subjects of the various courses (software engineering and distributed systems). However, it was not integral to the course material; i.e. we did not teach it in classes or lectures, nor did we expect the students to learn it for their respective courses, and it was not examinable.

Thus, it made sense for us to offer students participating in the study bonus points rather than standard course points, also ensuring that we were not disadvantaging those students who opted not to participate, as they could still earn the full standard course points. It was also possible for students to receive bonus points by completing the experiment tasks, but to opt out of having their contributions included in the experiment.

We pseudonymised participants' submissions after each experiment session so their identities did not influence the evaluation of their submissions. Similarly, their participation, non-participation or opting out of including their submissions in the experiment results did not affect their grading, except for the awarding or non-awarding of bonus points at the end of the semester.

[5] https://ufind.univie.ac.at/en/course.html?lv=052500&semester=2022S
[6] https://ufind.univie.ac.at/en/course.html?lv=052500&semester=2022W
[7] https://ufind.univie.ac.at/en/course.html?lv=053020&semester=2022S
[8] https://ufind.univie.ac.at/en/course.html?lv=051050&semester=2022W

Please note that a vote by the University of Vienna's Ethics Committee was not necessary according to their guidelines, as our study could not threaten the research subjects' physical and psychological integrity, the right to privacy was preserved through complete anonymisation through double-blinding (we used a procedure that was checked by the University of Vienna's data protection officer) and the participants could take part in the bonus point activity of the class but opt out of the experiment without any possible negative consequences. Due to the double-anonymisation, neither the course instructors nor the authors of the study knew who had opted out.

Including participants from courses at different stages of degree programmes enabled us to use them as proxies for novice (SE2), novice-to-moderately advanced (DSE) and moderately advanced (ASE) software architects, designers or developers. Whilst our study did not specifically involve software architects or ML engineers, the demographic data we collected suggests that the students who participated in our experiment could effectively represent professional software developers. These students possess a broad spectrum of knowledge and experience in software development. To illustrate, when we compare their demographics to those of a 2016 survey that encompassed 50,000 developers on the online platform Stack Overflow[9], we observe similarities in the most pertinent demographics compared to our participants, particularly that all of our participants had some degree of programming experience (please refer to Section 6.5.2).

Kitchenham et al. [84] provide further justification for using students in research experiments and state that using student subjects "*is not a major issue as long as you are interested in evaluating the use of a technique by novice or non-expert software engineers*" and that "*students are the next generation of software professionals and so are relatively close to the population of interest*". Furthermore, as noted in Höst et al. [180], Runeson [181], Svahnberg et al. [182], Salman et al. [183] and Falessi et al. [184], students may also represent professionals in empirical software engineering studies. We further discuss the use of students in our study in Section 6.6.2.

6.3.7 Material and Tasks

Material

We based the experiment on a selection of publicly available MLOps system descriptions, which we described in Section 6.3.2. The systems and subject matter were relevant to the topics covered in SE2, DSE and ASE but not formally part of the course content. The study consisted of the following material[10]:

- An *experiment information document*, providing background information on MLOps concepts and informal MLOps system descriptions and diagrams, and an example system description.
- An *experiment document* containing a survey requesting participant background information such as age, gender, education level, programming and industry ex-

[9] `https://survey.stackoverflow.co/2016`

[10] All relevant artefacts are available in our replication package [185].

perience, three informal textual system descriptions and corresponding informal MLOps system diagrams with associated tasks (described below), task surveys (also described below) and a concluding survey (which we did not use in the final evaluation). We released this document to participants at the start of their experiment sessions. A brief excerpt from an informal textual system description follows[11]:

> **Release pipeline**
>
> This pipeline shows how to operationalize and promote the scoring image safely across different environments. This pipeline is subdivided into two environments, QA and production:
>
> **QA environment**
>
> – **Model Artifact trigger.** Release pipelines get triggered every time a new artifact is available. A new model registered to Azure Machine Learning Model Management is treated as a release artifact. In this case, a pipeline is triggered for each new model that is registered.
>
> – **Create a scoring image.** The registered model is packaged together with a scoring script and Python dependencies (**Conda YAML file**) into an operationalization Docker image. The image automatically gets versioned through Azure Container Registry.
>
> – **Deploy on Container Instances.** This service is used to create a non-production environment. The scoring image is also deployed here, and this is mostly used for testing. Container Instances provides an easy and quick way to test the Docker image.
>
> – **Test web service.** A simple API test makes sure the image is successfully deployed.
>
> **Production environment**
>
> – **Deploy on Azure Kubernetes Service.** This service is used for deploying a scoring image as a web service at scale in a production environment.
>
> – **Test web service.** A simple API test makes sure the image is successfully deployed.
>
> The release pipeline publishes a real-time scoring web service. A release to the QA environment is done using Container Instances for convenience,

[11] We took the excerpt from the following source, which provides the full text and a corresponding informal MLOps system diagram: `https://tinyurl.com/mlops-system-ma`

> but you can use another Kubernetes cluster running in the QA/staging environment.

- We provided Experimental with supplementary information, including a description of how to understand semi-formal MLOps system diagrams with reference to UML-based generic component and pipeline diagrams. We also provided participants in Experimental with UML-based semi-formal MLOps system component and pipeline diagrams for each of the three systems, which are described in Section 6.3.2, and examples of a component and pipeline diagram for one of the systems may be found in Figures 6.3 and 6.4.

We provided two types of semi-formal, UML-based MLOps architecture diagrams (component and pipeline diagrams), rather than a single diagram, as they capture the high-level constructs identified in our prior empirical studies of ADDs for the ML workflow and deployment [23], [24]. We automatically annotated the UML diagrams with behaviour and properties using stereotypes and connectors. We generated them from a Python-based model, similar to how we generated the ADD diagrams in those studies.

As discussed in Section 6.2.1, MLOps systems are inherently intricate, often characterised by high complexity due to the integration of ML models, data pipelines, deployment processes and other components. Semi-formal UML-based MLOps systems diagrams, such as those provided in the experiment document and depicted in Figures 6.3 and 6.4, are invaluable tools for breaking down this complexity into more manageable components and are pivotal in enhancing understanding of MLOps-specific challenges.

Figure 6.3 depicts the components for the Microsoft Azure DevOps and Azure Machine Learning system described in Section 6.3.2. The Azure Blob Storage component, which stores various types of data, can trigger either the Azure Machine Learning Pipeline Endpoint via an API call (which subsequently triggers the Azure Machine Learning Retraining Pipeline) or the Azure DevOps Build Pipeline (a CI pipeline orchestration component).

The build pipeline provides an ML pipeline to the Azure Machine Learning Retraining Pipeline, which is an orchestration component type, and subsequently provides an ML model to the Azure Machine Learning Model Management model registry component. This registry then triggers the Azure DevOps Release Pipeline. This deployment pipeline orchestration component type deploys model images to Azure Container Instances (for staging or quality assurance environments) or Azure Kubernetes Services (for production environments).

Azure App Insights is a performance monitoring component that tracks the model's performance in Azure Kubernetes Service. The Azure Kubernetes Service saves model performance data to Azure Blob Storage.

Figure 6.4 depicts the build pipeline for the Microsoft Azure DevOps and Azure Machine Learning system described in Section 6.3.2. The build pipeline operates as a crucial component of the development process, triggered automatically with each code check-in. This pipeline's primary function is to publish an updated Azure Machine Learning pipeline. This process encompasses building the code and executing a suite of tests to ensure the integrity and quality of the codebase. The build pipeline itself

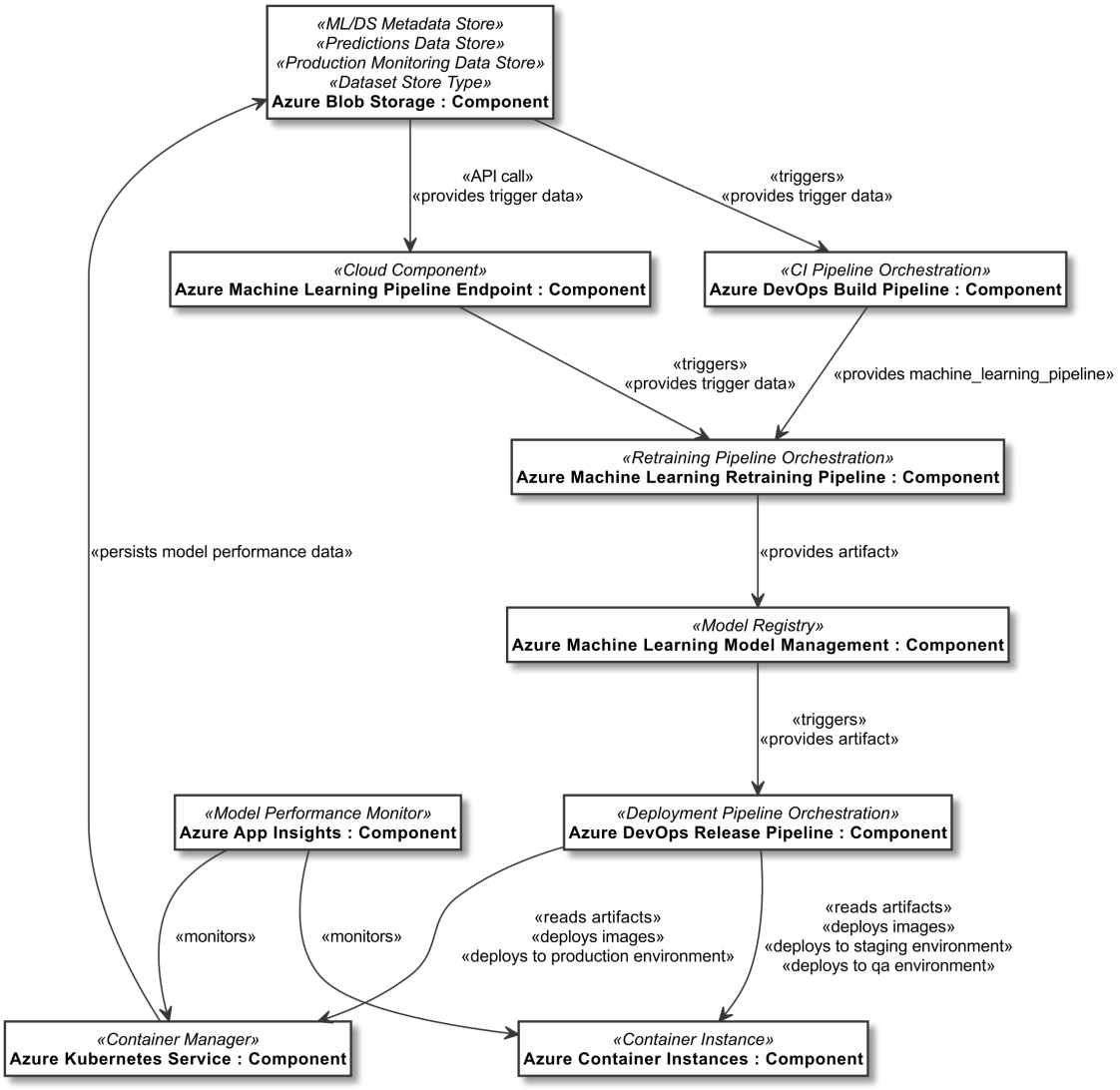

Figure 6.3: Microsoft Azure DevOps and Azure Machine Learning component diagram.

consists of several distinct steps, each contributing to the overall goal. All pipeline steps run automatically within the Azure DevOps Build Pipeline. The test steps take their input from Azure Blob Storage, and all pipeline steps run in either Azure Pipelines or Azure Machine Learning environments.

One key aspect of the build pipeline focuses on code quality. These tasks are instrumental in ensuring the code adheres to the development team's established standards and guidelines. Unit tests play a vital role in verifying code functionality, assessing code coverage and confirming stability. Additionally, the build pipeline encompasses data tests. These tests are specifically designed to verify that the data samples utilised within the system align with the anticipated schema and distribution. It is worth noting that the

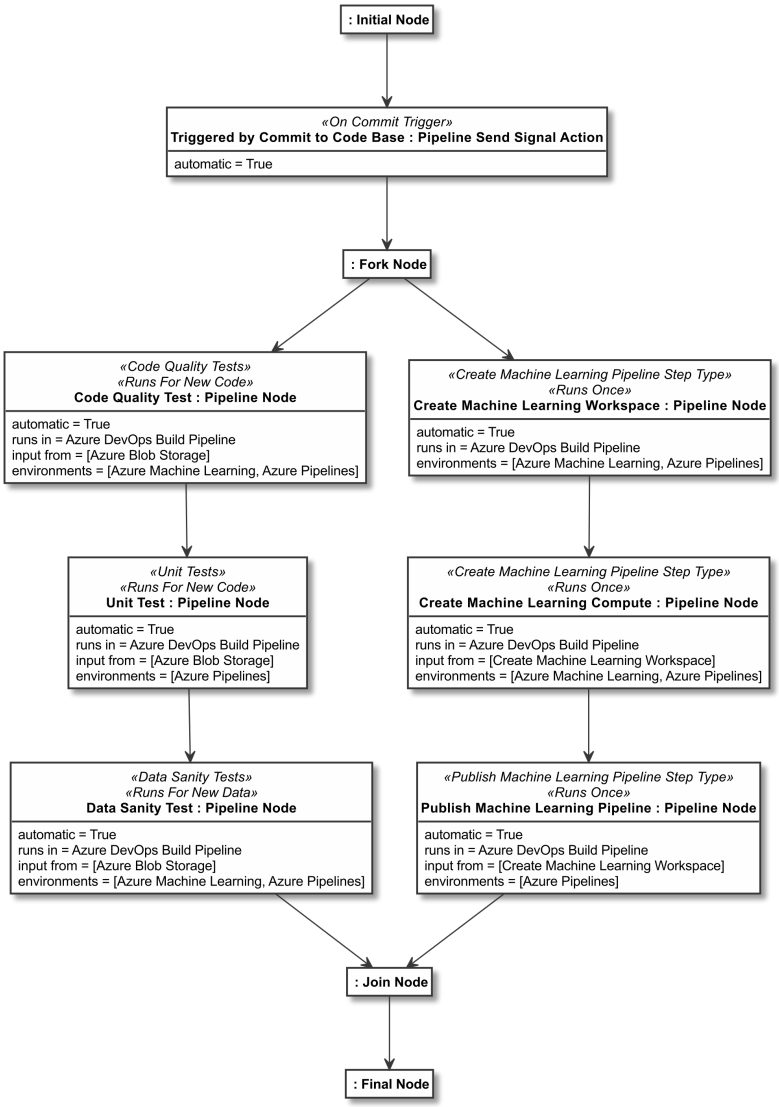

Figure 6.4: Microsoft Azure DevOps and Azure Machine Learning build pipeline diagram.

data test can be customised to suit different use cases.

Aside from the recurring build pipeline tasks, there are several one-time setup tasks associated with establishing the infrastructure for Azure Machine Learning and the Python SDK. These tasks are foundational to the ML system's overall operation. Firstly, creating the workspace that hosts all Azure Machine Learning-related resources is a critical initial step. The workspace serves as the central hub for organising and managing various ML components and resources. Another essential task is setting up the computing resources that execute training jobs. These resources are pivotal for the efficient execution of ML

tasks, ensuring that the necessary computational power is readily available. Subsequently, creating the ML pipeline, complete with the updated training script, is a one-time task that streamlines the workflow. This pipeline encapsulates the entire process of training ML models and orchestrating the associated tasks. Finally, the ML pipeline is published as a REST endpoint, providing a valuable interface for orchestrating the training workflow. This REST endpoint simplifies the initiation and management of training runs, promoting a more streamlined, automated approach to ML.

One notable benefit of UML-based MLOps system diagrams is that they can be designed to emphasise best practices and principles specific to MLOps. They serve as a visual guide to ensure the system is implemented consistently with industry standards. This alignment with established practices, including those presented in our previous work [23], [24], enhances the system's efficiency and reliability.

These diagrams incorporate annotations or labels that highlight known anti-patterns, such as those described by Sculley et al. [38], bringing them to the forefront. Practitioners can promptly recognise anti-patterns in the system's design, thereby simplifying the identification and addressing of problematic structural elements. When visualised within the broader system context, anti-patterns become more apparent, helping practitioners understand their implications and address them effectively.

As in other areas of software engineering, the use of semi-formal representations has been shown to improve understanding (see Stevanetic et al. [186] and Haitzer and Zdun [187]) and we speculate that ML practitioners may also benefit when utilising semi-formal representations, for instance, to identify areas where technical debt (see Sculley et al. [38]) may accumulate within the system. These diagrams provide a visual guide to software modules and their dependencies, making it easier to recognise segments that may require refactoring or enhancements. Annotations or metadata within diagrams can highlight potential sources of technical debt, providing a visual cue for practitioners to focus on addressing them effectively.

Lastly, semi-formal MLOps system architecture view descriptions serve as instrumental tools for illustrating the use of ML models, including strategies for model selection, storage, versioning and reuse. This visual representation provides practitioners with insights into the criteria and processes that guide model choices and their integration into the system. Practitioners can discern how models interact with various components, thereby facilitating their comprehension of the design and decision-making processes involved in model selection and reuse.

Tasks

We associated each of the three systems with three tasks relating to aspects common to MLOps systems, to be answered after reading and understanding each system description. An example of each task type follows.

Pipeline behaviour tasks A "select the true statements"-style question with four statements to choose from, for example:

Please keep time records and tick (✓) the true statements:

- ☐ The retraining pipeline always finishes by registering a model.
- ☐ The retraining pipeline can be triggered via a suitable API call to an appropriate component.
- ☐ The build pipeline can trigger the retraining pipeline.
- ☐ In the release pipeline, releases of images to production can be approved independently of whether the images were first deployed to staging and QA.

Components tasks A "list all components..."-style question with four different types of components to list, for example:

Please keep time records and list all components that...

- ...are cloud components: _____ _____.
- ...are orchestration components: _____ _____.
- ...provide an artefact to another component: _____ _____.
- ...receive trigger data from another component: _____ _____.

Pipeline properties tasks An "enter the correct number"-style question, for example:

Please keep time records and enter the correct number for each of the following statements:

- The various pipelines can be triggered by _____ type(s) of trigger in total.
- The pipelines contain _____ node(s) responsible for testing.
- The pipelines contain _____ step(s) that do not run automatically.
- The total number of pipelines triggered via a commit to the code base is _____.

The task design in the controlled experiment, encompassing questions related to pipeline behaviour, components and pipeline properties, is based on findings from our previous grey literature studies [23], [24] on ADDs, patterns and practices for ML. Technical components and their related pipelines are important in various MLOps practitioner roles for task

execution and the successful implementation of MLOps principles such as automation, workflow orchestration and reproducibility, as discussed by Kreuzberger et al. [4]. Thus, we consider our study tasks relevant to the factors necessary for the decision-making processes regularly undertaken by practitioners, particularly those engaged in MLOps, as part of their routine work. These tasks serve as a bridge between the high-level architectural understanding and the low-level implementation details, rendering them highly pertinent to real-world engineering practice.

Practitioner sources from our prior studies [23], [24] described earlier in this doctoral thesis, and others, such as Lakshmanan et al. [1] and Treveil et al. [6], highlight the importance and relevance of pipelines to MLOps. Unlike traditional CI/CD setups, such as those described by Humble and Farley [22] and Shahin et al. [165], MLOps architectures may consist of a wider variety of interconnected, fundamental, ML-specific pipeline types and steps, such as data preprocessing pipelines, model training and evaluation pipelines, data ingestion steps, feature engineering steps and hyperparameter tuning steps, as well as pipeline trigger types for new training data availability, model performance degradation and changes in the data distribution. We argue that our tasks on pipeline behaviour facilitate the assessment of participants' understanding of the functioning of MLOps systems in their pipelines. They shed light, for instance, on how pipelines register models, trigger other pipelines and release images to different environments, thereby ensuring the system's correct and reliable operation. This knowledge closely aligns with the insights practitioners, particularly those specialising in MLOps, need in real-world contexts.

Counting, for instance, different trigger types, testing nodes, non-automatic steps or pipelines initiated through code commits mirrors the evaluation of specific properties within the system. We regard such evaluations as common practice amongst developers and consider them integral to understanding system behaviour when identifying bottlenecks and ensuring accurate pipeline configuration. The tasks correlate directly with the decisions developers make when configuring and automating various facets of MLOps systems. For instance, determining how pipelines are triggered and specifying the actions they should take is a practical necessity for building reliable CI/CD pipelines.

Quantifying the number of specific properties within the pipelines aligns with the practical need to evaluate them for quality assurance and system performance. Developers routinely perform such assessments to unearth areas for improvement or potential issues.

According to Humble and Farley [22], almost all modern software systems consist of a collection of components, and the relations between them in build and deployment processes can influence a system's complexity. They state that considering their interactions when implementing a deployment pipeline is challenging. The relations they describe between components and pipelines are also relevant for MLOps, for instance, how a deployment pipeline interacts with an artefact repository component or the level of interaction between a version control system and build infrastructure components. On this basis, we regard understanding MLOps components and their relationships as important for MLOps practitioners and our study task involving identifying components and their interactions should help assess how well participants understand relevant aspects of the system architectures described in the experiment.

Identifying distinct component types and articulating their roles mirrors the real-world decisions developers frequently make when selecting and integrating components within a system. This identification process is crucial to ensuring that the chosen components work harmoniously together.

The assigned tasks parallel high-level architectural analysis, a domain in which developers strive to understand the general structure and behaviour of MLOps systems. Questions about the understanding and identification of pipeline behaviours and components are, in our considered opinion, well-suited to assessing comprehension of a system's overarching architecture. Concurrently, these tasks delve into the minutiae by considering properties such as the number of trigger types and the number of steps that do not run automatically. In our view, analysing such low-level details is pivotal in system implementation and fine-tuning.

Tasks for both groups were identical. Each task concluded with a SELF-ASSESSMENT survey on a Likert scale, representing a subjective evaluation of each participant's performance. The survey questions, to be answered on a scale from "strongly agree" to "strongly disagree", were as follows:

- "I am confident that my provided answers and solutions for this task are correct."
- "It was easy for me to understand the textual system description."
- "It was easy for me to understand the system diagram(s)."
- "It was easy for me to identify pipelines."
- "It was easy for me to identify components."
- "It was easy for me to understand pipeline behaviour."
- "It was easy for me to understand components and their relations."
- "It was easy for me to understand properties of pipelines."

6.3.8 Variables, Hypotheses and Research Question

The independent variable is the provision or non-provision of the supplementary material (semi-formal MLOps system diagrams). Two dependent variables are defined: the CORRECTNESS of the answers provided to the tasks and the DURATION, i.e. the time taken to complete the tasks.

The two dependent variables, particularly the CORRECTNESS, may be used to gain insight into the overall understandability of system descriptions [158], [159], [160], [188], [189]. In the context of our experiment, CORRECTNESS measures whether participants correctly identify and understand the components, pipelines, architectural roles, behaviour, properties and relations of the described software architecture. A high level of CORRECTNESS indicates that the architecture is easy to understand and navigate. By contrast, a low level of CORRECTNESS suggests that the architecture is more complex and challenging to comprehend. DURATION measures the time it takes participants to

complete the tasks assigned to them in the experiment. A shorter DURATION indicates that participants can quickly and efficiently understand the software architecture, whereas a longer DURATION suggests that the architecture is more challenging to comprehend.

We hypothesised that MLOps system descriptions are easier to understand when semi-formal MLOps system diagrams are provided alongside informal textual and MLOps system diagrams. Thus, for *understanding*, we defined the following null and corresponding alternative hypotheses:

- $\mathbf{H_0 1}$: There is no significant difference in task CORRECTNESS when semi-formal MLOps system diagrams, in addition to informal textual descriptions and informal graphical system diagrams, are provided.
- $\mathbf{H_a 1}$: Task CORRECTNESS increases significantly when semi-formal MLOps system diagrams, in addition to informal textual descriptions and informal graphical system diagrams, are provided.

Furthermore, we wished to investigate the effects of providing supplementary material in the form of semi-formal MLOps system diagrams to EXPERIMENTAL on task DURATION. We formulated the following null and alternative hypotheses accordingly:

- $\mathbf{H_0 2}$: There is no significant difference in task DURATION when semi-formal MLOps system diagrams, in addition to informal textual descriptions and informal graphical system diagrams, are provided.
- $\mathbf{H_a 2}$: Task DURATION increases significantly when semi-formal MLOps system diagrams, in addition to informal textual descriptions and informal graphical system diagrams, are provided.

Finally, we wished to investigate any relationship between task DURATION and task CORRECTNESS and thus formulated the following null and alternative hypotheses:

- $\mathbf{H_0 3}$: There is no significant increase in task CORRECTNESS as task DURATION increases when only informal textual descriptions and informal graphical system diagrams are provided.
- $\mathbf{H_a 3}$: Task CORRECTNESS increases significantly as task DURATION increases when only informal textual descriptions and informal graphical system diagrams are provided.
- $\mathbf{H_0 4}$: There is no significant increase in task CORRECTNESS as task DURATION increases when semi-formal MLOps system diagrams, in addition to informal textual descriptions and informal graphical system diagrams, are provided.
- $\mathbf{H_a 4}$: Task CORRECTNESS increases significantly as task DURATION increases when semi-formal MLOps system diagrams, in addition to informal textual descriptions and informal graphical system diagrams, are provided.

This chapter is based on **P5**, corresponds to **RC5** and addresses **RP3**. We defined a suitable research question as part of **RQ3** to discover how confident participants of each group were in their performance:

RQ3.1 *How does the exclusion or inclusion of semi-formal MLOps system diagrams affect the accuracy with which participants assess their performance in the context of their perceived task correctness?*

6.4 Experiment Execution

6.4.1 Pilot Test

After preparing our experiment materials, we conducted preliminary tests involving student tutors from our research group. Much like the participants in the real experiment, we provided the tutors with the information sheet two weeks in advance, allowing them to familiarise themselves with the necessary knowledge to address the tasks. We randomly allocated one tutor to each of the Control and Experimental groups. They engaged in the experiment under the same conditions we had planned for the official experiment sessions. These conditions included the option to seek clarifications regarding the experimental procedure, a 75-minute time frame for responding to the experiment questions, and no supplementary support beyond the provided materials.

After the pilot tests, we sought feedback from the tutors regarding their experiences. Their responses indicated that we had crafted the information sheet exceptionally well and provided ample guidance for addressing the experiment questions. They found the experiment to be straightforward to navigate. They noted that even participants new to the subject matter and concepts would likely find it accessible, primarily due to the comprehensive materials and examples provided. Given this positive feedback, we made no alterations to our experiment design.

6.4.2 Preparation

To help participants prepare for their experiment sessions, we provided registered participants with the information document described in Section 6.3.7 via Moodle, our e-learning platform, two weeks before their session. The purpose of providing the information sheet was to ensure that participants had the same minimum prerequisite knowledge and understanding of the subject matter, thereby reducing the in-session learning curve and minimising potential bias amongst participants due to factors such as education level, prior knowledge and programming or industry experience. All participants worked through three tasks per described MLOps system.

Once we had concluded all experiment sessions, we consolidated the collected data and derived and analysed descriptive statistics. The analysis aimed to determine whether the statistics supported or refuted the hypotheses defined in Section 6.3.8.

At various stages of the experiment, we also asked participants to complete brief surveys to assess their self-perception of performance, using a Likert scale. We used

these survey responses to gain insight into how well participants thought they had performed compared to how well they actually performed (in terms of CORRECTNESS) and to answer our research question.

6.4.3 Execution

We conducted the experiment sessions as a pen-and-paper exercise under controlled conditions, similar to a traditional closed-book written examination, with no supplementary tools or resources permitted, except for the information sheets distributed in advance. To ensure equal access to the supplementary information, we provided copies of the sheets in case participants did not bring their own. We set an upper limit on DURATION that was the same for all participants, we did not allow collaboration between participants, we alternately assigned participants to either CONTROL or EXPERIMENTAL, and we ensured that participants sat at a sufficient distance from one another to prevent collaboration or copying.

After providing a 15-minute overview of the experimental procedure and the structure of the experiment sheets at the start of the session, we issued each participant an experiment sheet associated with either EXPERIMENTAL or CONTROL. We instructed participants to enter their background information and complete the tasks and embedded surveys in consecutive order. We allowed the participants a maximum of 75 minutes to complete the tasks and required them to record the time intervals spent reading the system descriptions and completing tasks. We projected a clock with second-level granularity onto a screen to facilitate participants' timestamp recording.

6.5 Analysis

6.5.1 Dataset Preparation

We could not include all participants' contributions in the study, and defined objective exclusion criteria as follows:

- Participants recorded timestamps but made systematic errors throughout the experiment, so we could not infer the correct timestamp values ($n = 1$). This single participant, who was excluded for this reason, did not record timestamps correctly, recording them to whole minutes rather than whole seconds.

- Participants systematically recorded timestamps throughout the experiment that led to obviously implausible DURATION values ($n = 1$). The single participant excluded for this reason recorded taking 4 seconds to read the supplementary material, 7-13 seconds to read the system descriptions and 0-5 seconds to complete the tasks. In contrast, most other participants would typically take several minutes for these activities.

- Participants did not confirm that they had read and understood the information sheet before the experiment session ($n = 3$). Confirmation involved ticking the

relevant checkbox on the experiment document's initial survey page.

- Participants did not consent to their contribution being used in the study ($n = 1$). This confirmation also involved ticking the relevant checkbox on the experiment document's initial survey page. The single participant who opted out of having their contribution included did not confirm that they had read and understood the information sheet before the experiment session either. We would have excluded this student anyway for that reason, and we included them in the count of the excluded students above.

- Students could participate in both semesters to gain extra credit in multiple courses. However, for repeated participants, we included only the contribution from the first experiment session in which each student participated ($n = 4$).

Thus, we excluded nine of 72 contributions, leaving 63 unique contributions for inclusion. Of the included contributions, we conducted some necessary further manual processing:

- If an individual timestamp for a task was missing but could be inferred, then we used the inferred value.

- If at least one timestamp for a task was missing and could not be inferred, then we excluded the time for this task.

- If a task had not been attempted, we excluded it.

- If a survey response was not completed, then we excluded this survey response from the overall results. However, we still included the remaining survey responses (excluding any other incomplete responses).

- One participant misunderstood a task and communicated this in writing on the experiment sheet. We excluded their contribution for this individual task from the results but still included the remainder of their contribution (barring any other excluded tasks).

The implications of excluding individual tasks and survey responses are as follows: if we excluded a task for any reason, then we recorded no score for the task and did not use it when calculating the average CORRECTNESS, nor did we use it when calculating the total DURATION; if a survey response was missing, then we did not use it to gain insight into how well participants thought they had performed compared to how well they actually performed.

We entered the data from the contributions to be included in the study, including the participants' background information, survey responses, time intervals and task answers, into a LibreOffice [190] OpenDocument Spreadsheet (ODS)[12] file. We converted the participants' start and stop timestamps into a DURATION in seconds.

[12] https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=office

We exported the ODS file to a comma-separated value (CSV) [191] file and, having already installed the necessary R [173] dependencies (we used the R programming language for the statistical analysis), we then ran the CSV file through an R script, which performed initial processing and generated an RDS [192] file. We input the RDS file into a different R script, which analysed the data and generated plots and tables of statistics.

6.5.2 Participant Demographics

Participants' background information that we collected included age[13] (see Figure 6.5), gender, course, education level, programming experience (see Figure 6.6), modelling experience, software industry experience (see Figure 6.7), hardware industry experience and programming and modelling language experience. We also collected data on whether participants had prior knowledge of ML, CI/CD and CI/CD for ML. Neither group appeared to have an unfair advantage over the other, and any imbalances for specific demographics were compensated elsewhere.

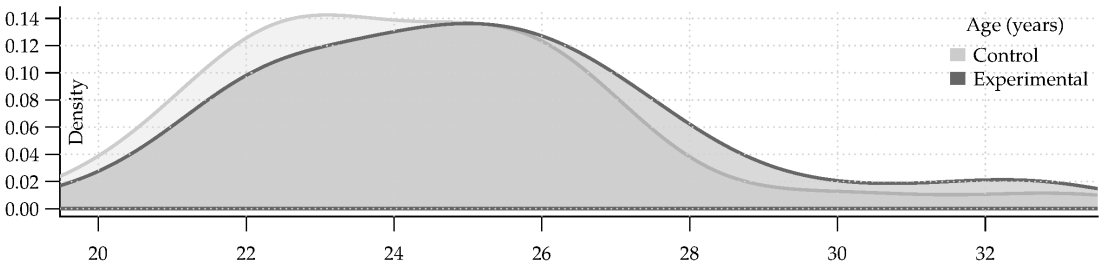

Figure 6.5: Kernel density plot of participants' ages.

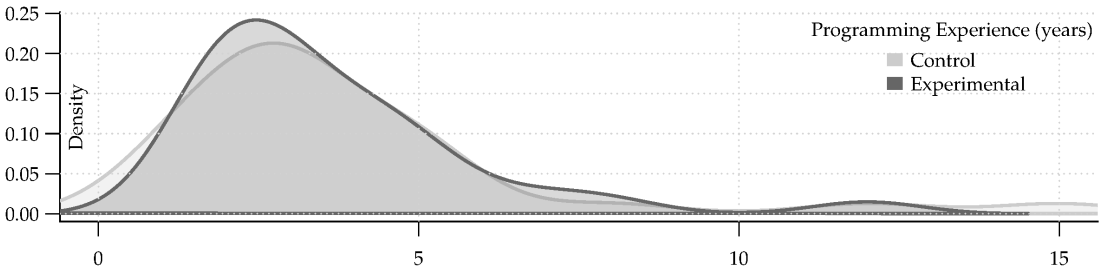

Figure 6.6: Kernel density plot of participants' programming experience.

[13] Two participants did not provide their ages.

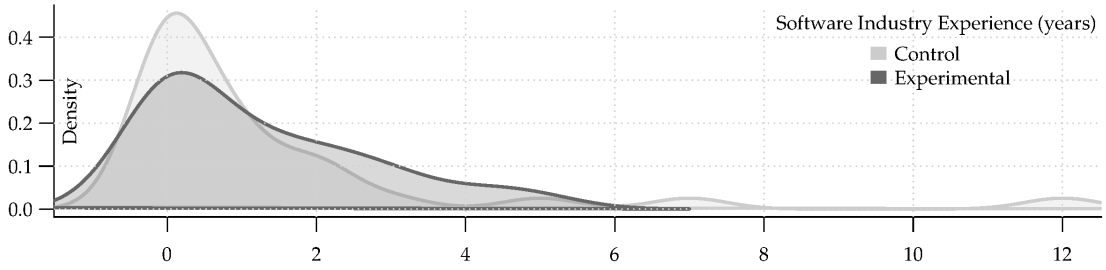

Figure 6.7: Kernel density plot of participants' software industry experience.

Of the 63 participants, 34 had no higher education qualifications ($n_{\text{CONTROL}} = 15$, $n_{\text{EXPERIMENTAL}} = 19$)[14], 26 had a bachelor's degree ($n_{\text{CONTROL}} = 14$, $n_{\text{EXPERIMENTAL}} = 12$) and three had a master's degree ($n_{\text{CONTROL}} = 2$, $n_{\text{EXPERIMENTAL}} = 1$).

Across both groups, 42 participants had prior knowledge of ML ($n_{\text{CONTROL}} = 19$, $n_{\text{EXPERIMENTAL}} = 23$), 34 had prior knowledge of CI/CD ($n_{\text{CONTROL}} = 13$, $n_{\text{EXPERIMENTAL}} = 21$), two had prior knowledge of CI/CD for ML ($n_{\text{CONTROL}} = 0$, $n_{\text{EXPERIMENTAL}} = 2$), 35 had prior knowledge of modelling ($n_{\text{CONTROL}} = 19$, $n_{\text{EXPERIMENTAL}} = 16$) and seven participants had no prior knowledge of any of the above ($n_{\text{CONTROL}} = 5$, $n_{\text{EXPERIMENTAL}} = 2$).

Regarding education level, CONTROL was slightly better qualified, with four fewer participants without a university education, two more with a bachelor's degree and one more with a master's degree. Regarding prior knowledge of CI/CD, EXPERIMENTAL appeared to be more experienced, since that group had eight more participants with prior knowledge of CI/CD and two more with prior knowledge of CI/CD for ML. On the other hand, CONTROL had three more participants with prior modelling knowledge. The average age in CONTROL was 24.43, and in EXPERIMENTAL, 25.16, i.e. roughly the same. The mean number of years of experience in the software industry was very similar, with 1.34 for CONTROL and 1.25 for EXPERIMENTAL. The number of years of programming experience was also comparable across groups, with CONTROL averaging 3.85 years and EXPERIMENTAL averaging 3.62 years. Thus, neither group had an overall advantage over the other in multiple demographic aspects, and any advantage one group may have had in one area was offset by a different area for the other group.

6.5.3 Descriptive Statistics

In Table 6.1, we note the group-specific statistics for the dependent variables CORRECTNESS and DURATION, which we define within the range $[0, 1] \cap \mathbb{R}$. This table presents the number of observations along with measures of central tendency and dispersion for each group.

We utilise various data representations in this chapter, each providing valuable insights and informing our choice of the most suitable statistical tests. Kernel density plots provide insights into the distribution, whilst Q-Q plots assess normality. The box plots

[14] n_{CONTROL}. is the number of observations in CONTROL and $n_{\text{EXPERIMENTAL}}$ is the number of observations in EXPERIMENTAL.

are used for summary statistics, the scatter plots reveal potential correlations between two dependent variables, and the tables display a variety of descriptive statistics.

Table 6.1: Descriptive Statistics per Group of Dependent Variables Correctness and Duration

Correctness		
	Control	Experimental
Observations	31	32
Mean	0.3553	0.6947
Standard deviation	0.0721	0.1643
Median	0.3676	0.7183
Median abs. deviation	0.0728	0.1431
Minimum	0.2083	0.2292
Maximum	0.4884	0.9167
Skew	-0.3261	-1.0567
Kurtosis	-0.8388	0.4515
Shapiro-Wilk Test p	0.4359	0.0056

Duration		
	Control	Experimental
Observations	24	25
Mean	1806.88	1861.56
Standard deviation	621.19	478.73
Median	1916.50	1994
Median abs. deviation	436.63	416.61
Minimum	744	668
Maximum	2989	2523
Skew	-0.1642	-0.7997
Kurtosis	-0.9032	-0.2763
Shapiro-Wilk Test p	0.2826	0.0711

Correctness Descriptive statistics for correctness can be viewed as a kernel density plot presented in Figure 6.8, as well as a box plot provided in Figure 6.9 and a Q-Q plot in Figure 6.10.

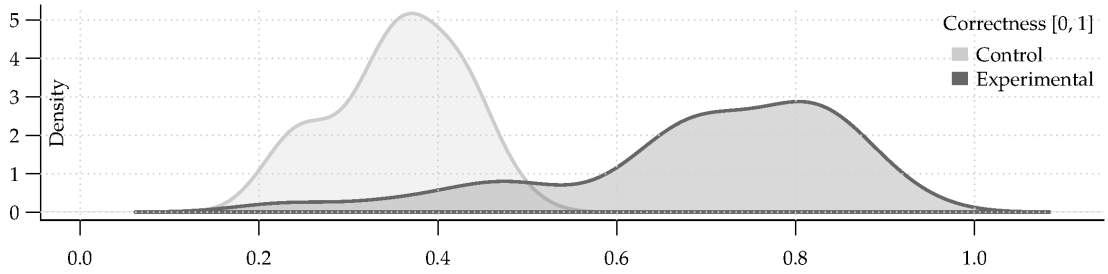

Figure 6.8: Kernel density plot of CORRECTNESS.

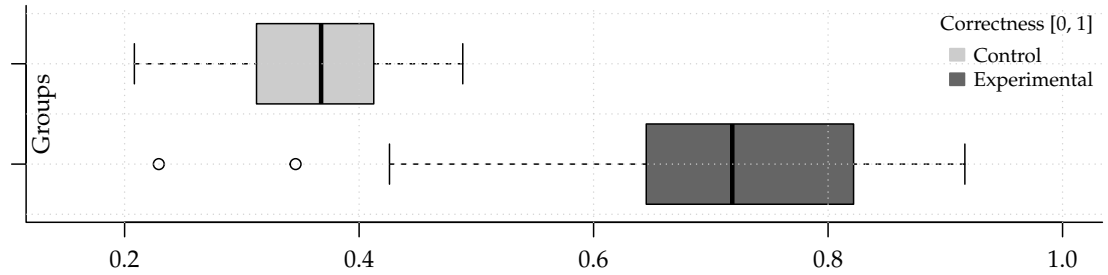

Figure 6.9: Box plot of CORRECTNESS.

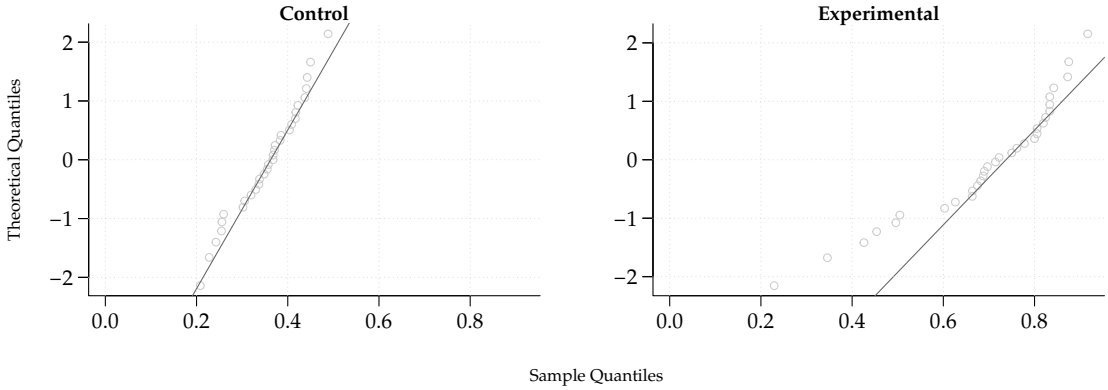

Figure 6.10: Normal Q-Q plot of CORRECTNESS.

In the box plot displayed in Figure 6.9, it is clearly noticeable that the median and the entire interquartile range of EXPERIMENTAL exceed those of CONTROL and even surpass the maximum value of CONTROL. The boxes, which do not overlap, emphasise the difference between the two groups in terms of CORRECTNESS. Furthermore, EXPERIMENTAL features an extended lower whisker due to some participants' underperformance compared to other group members. Additionally, EXPERIMENTAL exhibits two outliers with notably

low CORRECTNESS values[15]. Upon visually examining Figure 6.8 and considering the skew values provided in Table 6.1, it becomes evident that the distribution of CONTROL closely approximates a nearly symmetrical or lightly negatively skewed form. In contrast, EXPERIMENTAL's distribution displays a highly negative skew. The kurtosis values, both < 3, suggest that the distributions in both groups are platykurtic.

Examining the normal Q-Q plots in Figure 6.10 for both groups and CORRECTNESS, it appears that the data from CONTROL exhibits a visual resemblance to a normally distributed pattern. However, the normality of EXPERIMENTAL's data remains inconclusive based on these visual cues. To test for normality, we chose the Shapiro-Wilk [193] normality test, as it is more robust than alternatives (such as Anderson-Darling [194], Lilliefors [195] and Kolmogorov-Smirnov [196]) according to Razali and Yap [197]. Assuming $\alpha = 0.05$, this test indicated that CONTROL's distribution is not significantly different from the normal distribution, with a value $p > \alpha$, whereas EXPERIMENTAL's distribution significantly deviates from the normal distribution, with a value $p \leq \alpha$.

Duration Table 6.1 shows the number of observations, central tendency and dispersion measures per group for the dependent variable DURATION[16] These statistics are visualised as a kernel density plot in Figure 6.11 and as a box plot in Figure 6.12.

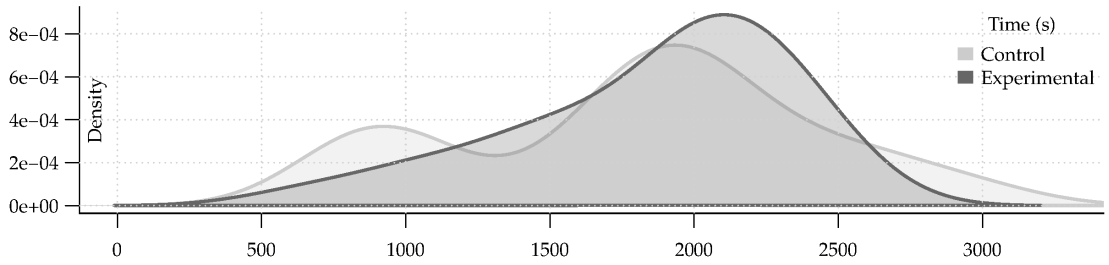

Figure 6.11: Kernel density plot of DURATION.

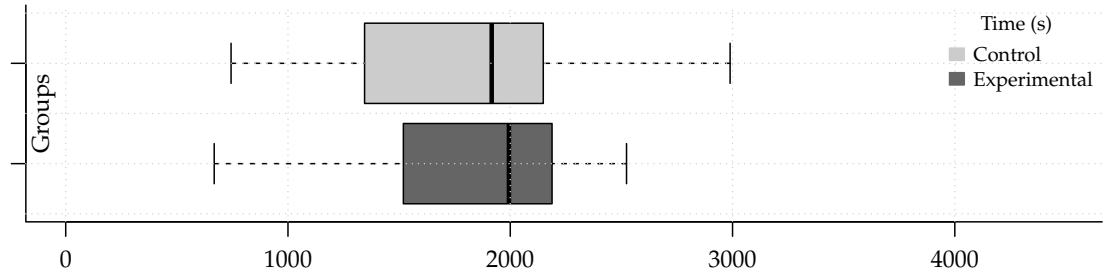

Figure 6.12: Box plot of DURATION.

[15] It is worth noting that we identified these outliers as natural variations within the population rather than arising from measurement, data entry or data processing errors, thus representing true outliers.

[16] We denote DURATION in seconds.

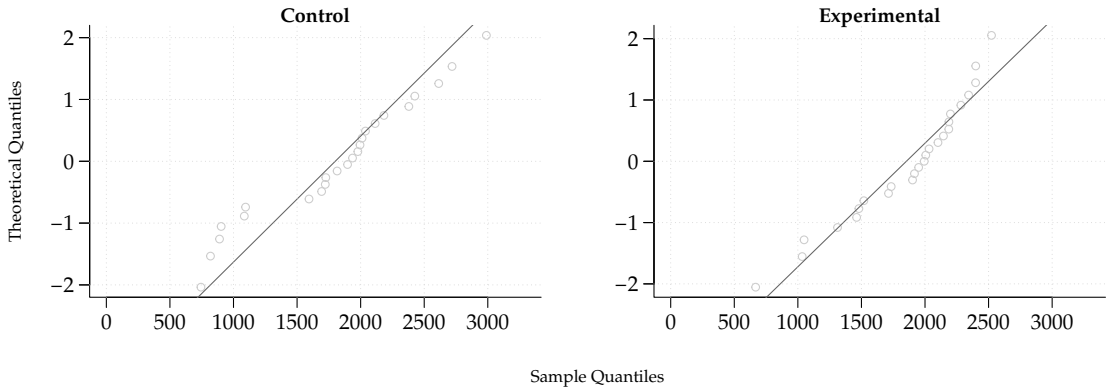

Figure 6.13: Normal Q-Q plot of DURATION.

The skews shown in Table 6.1 indicate that CONTROL's distribution is nearly symmetrical to lightly negatively skewed, whereas EXPERIMENTAL's distribution is moderately negatively skewed. The kurtosis values of < 3 for both groups indicate that the distributions of the groups are platykurtic.

In the box plot of Figure 6.12, the groups' interquartile ranges overlap, with EXPERIMENTAL's interquartile range almost being contained entirely within CONTROL's, with a slightly higher median. EXPERIMENTAL's box plot's whiskers indicate a smaller spread of values than CONTROL's, which is also confirmed by the lower standard deviation (478.73 for EXPERIMENTAL versus 621.19 for CONTROL). Whilst both groups' minimum values were similar, CONTROL's maximum value was relatively high, but we did not deem it an outlier for that group.

Visual inspection of the normal Q-Q plots for both groups of DURATION, shown in Figure 6.13, was insufficient to determine whether each group's data were normally distributed. The Shapiro-Wilk normality test indicated that, for DURATION, neither group's distribution is significantly different to normal, with values $p > \alpha$ for both groups.

6.5.4 Hypothesis Testing

Correctness and Duration When comparing two or more groups on two or more dependent variables and provided certain other assumptions are met, it is usually possible to apply the *Multivariate Analysis of Variance* (MANOVA) [198] statistical test to determine whether independent variables on their own affect dependent variables. As noted in Section 6.5.3, CONTROL's distribution is not significantly different from the normal distribution for CORRECTNESS and DURATION, whereas EXPERIMENTAL's distribution is significantly different from the normal distribution for CORRECTNESS but not for DURATION. Still, MANOVA requires all distributions to be normally distributed.

It was thus necessary to compare the variances of the two groups for each dependent variable to determine whether they were equal, as this would further inform our choice of statistical tests. We selected Bonett's [199] method as a two-sided test to compare variances. This test was a suitable choice since even though EXPERIMENTAL's CORRECT-

NESS data was heavily skewed, the sample sizes for all four combinations of group and variable were $n \geq 20$. Had the latter condition not been the case, then another test, such as Levene's [200] or Brown-Forsythe's [201] method, may have been more appropriate to reduce the risk of a high type I (false positive) error rate when data is highly skewed and $n < 20$. Bonett's test is also accurate for any continuous distribution of quantitative values and does not require normality. Given the above conditions, it is usually more powerful and reliable than Levene's or Brown-Forsythe's tests.

For CORRECTNESS, the estimated ratio is 2.143862, with a 95% confidence interval of $[1.401176, 3.280205]$ and $p = 0.0004406616$ with $\alpha = 0.05$. Since $p \leq \alpha$, the estimated ratio of the groups' standard deviations is statistically significant, i.e. the variances for CORRECTNESS differ between the groups. For DURATION, the estimated ratio is 0.76211, with a 95% confidence interval of $[0.4590661, 1.265203]$ and $p = 0.2935268$ with $\alpha = 0.05$. Since $p > \alpha$, the estimated ratio of the groups' standard deviations for DURATION is not statistically significant, i.e. the variances for DURATION do not differ.

Given that the dependent variable CORRECTNESS had differing variances between groups, we could not use the Kruskal-Wallis [202] test or Wilcoxon's [203] rank sum test [87], [204]. Welch's [205] t-test is suitable for unequal population variances but assumes normally distributed data, so this was not an option either.

Cliff's δ [206] is a robust, nonparametric test that makes no assumptions about data distribution, differing distributions between populations or unequal variances and is recommended as a suitable method in this scenario by Kitchenham [87]. Even though Cliff's δ was originally intended to measure ordinal data, it is equally applicable [207], [208] to the quantitative, continuous data in this study. Cliff's δ estimates the probability that a randomly selected observation from one group is larger than a randomly selected observation from a second group, subtracting the reverse probability [209].

When testing multiple hypotheses with a single method (we applied Cliff's δ twice), it is necessary to lower the level of α to avoid type I errors[17]. Various methods to adjust α can be chosen from, such as the false discovery rate [210] or the Bonferroni-Dunn [211], [212] correction. The latter is the strictest form of correction and is given by Equation 6.1:

$$\alpha' = \frac{\alpha}{n} \tag{6.1}$$

where n is the number of times a test was applied. In our study, this results in $\alpha' = \frac{0.05}{2} = 0.025$ where $\alpha = 0.05$ and $n = 2$.

The results of the one-tailed Cliff's δ test are shown in Table 6.2 for CORRECTNESS and DURATION. For CORRECTNESS, Cliff's δ indicates by $p \leq \alpha'$ that EXPERIMENTAL scored significantly higher than CONTROL. For DURATION, Cliff's δ yielded $p > \alpha'$, so we cannot conclude that EXPERIMENTAL took significantly longer than CONTROL to complete the experiment. To verify these results, we ran a one-tailed Brunner-Munzel [213] test for each dependent variable. This test is suitable for differing distributions and can also be applied to arbitrary data distributions. Adjusting α to derive α' for repeated tests via Bonferroni-Dunn correction as previously described, the results for the Brunner-Munzel

[17] Note that the α adjustment is not necessary when testing for normality or when comparing variances.

Table 6.2: Hypothesis Tests per Group Combination of the Dependent Variables Correctness and Duration

Correctness ($H_0 1$)	
	Experimental vs Control
Cliff's δ Test	
Cliff's δ	-0.8633
s_δ	0.0886
v_δ	0.0079
z_δ	-9.7401
CI_{low}	-0.9528
CI_{high}	-0.6359
$P(X > Y)$	0.9317
$P(X = Y)$	0.0000
$P(X < Y)$	0.0683
p	3.719e−13
Brunner-Munzel Test p	3.063e−10

Duration ($H_0 2$)	
	Experimental vs Control
Cliff's δ Test	
Cliff's δ	-0.0767
s_δ	0.1688
v_δ	0.0285
z_δ	-0.4542
CI_{low}	-0.3434
CI_{high}	0.2015
$P(X > Y)$	0.5383
$P(X = Y)$	0.0000
$P(X < Y)$	0.4617
p	0.3259
Brunner-Munzel Test p	0.3275

test as indicated in Table 6.2 align with those of Cliff's δ, with $p \leq \alpha'$ for correctness and $p > \alpha'$ for duration.

Based on both tests, concerning correctness, we can reject the null hypothesis $\mathbf{H_0 1}$, as the evidence supports the alternative hypothesis $\mathbf{H_a 1}$. We found a significant difference in task correctness in favour of providing semi-formal MLOps system diagrams alongside informal textual descriptions and informal graphical system diagrams. Conversely, concerning duration, we fail to reject the null hypothesis $\mathbf{H_0 2}$, since there is insufficient evidence against it. That is, we found no significant difference in task duration between Control and Experimental.

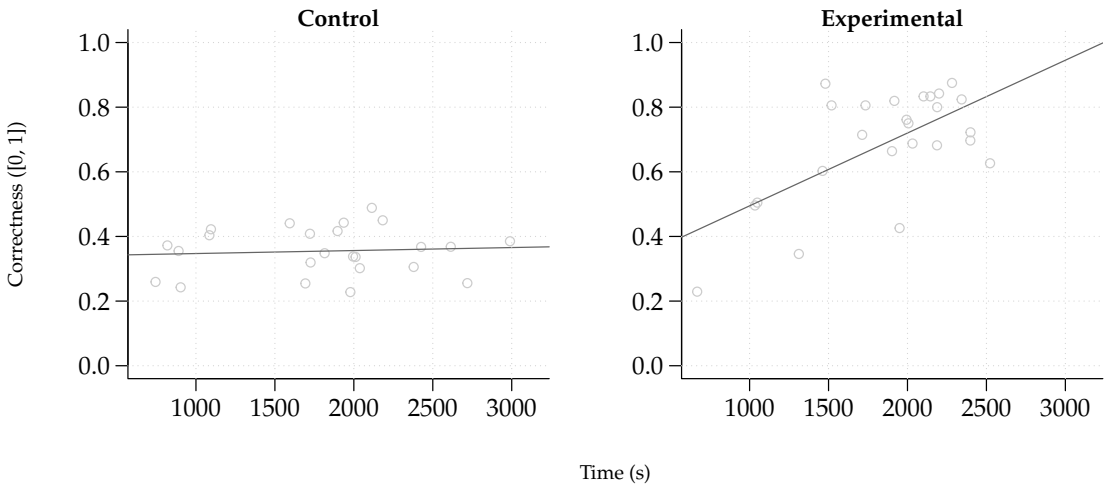

Figure 6.14: Scatter plot of the dependent variables CORRECTNESS to DURATION per group.

Correlation Between Correctness and Duration Visual inspection of the scatter plot depicting any potential correlation between the two dependent variables CORRECTNESS and DURATION in Figure 6.14 per group did not reveal an apparent linear correlation. For CONTROL, the CORRECTNESS barely increased with time. In the case of EXPERIMENTAL, despite an apparent increase in CORRECTNESS with DURATION, the data points lie so far from the indicated reference line that we cannot assume a linear correlation. Following this visual inspection, we deemed it prudent to test for correlation. Various tests were available, including three well-known ones: Pearson's correlation coefficient, Kendall's τ and Spearman's ρ.

The results of the tests we conducted are shown in Table 6.3. Pearson's [214] correlation coefficient is suitable for continuous variables that are both normally distributed. We established earlier in this section that this is not the case for EXPERIMENTAL's CORRECTNESS data; thus, we disregarded Pearson's correlation coefficient. The Kendall [215] rank

Table 6.3: Correlation per Group of the Dependant Variables CORRECTNESS with DURATION per Group

	CONTROL ($H_0 3$)	EXPERIMENTAL ($H_0 4$)
Kendall's τ	0.0580	0.3105
p	0.3457	0.0149
z	0.3969	2.1726
Spearman's ρ	0.0504	0.4339
p	0.4075	0.0151
S	2184	1471.7830

correlation coefficient (Kendall's τ) measures the relationship between two variables. It is

suitable for our purposes because it can handle continuous data and is nonparametric. We calculated the one-tailed Kendall rank correlation coefficient. For confirmation, we also calculated the one-tailed Spearman [216] rank correlation coefficient (Spearman's ρ), which is similar to Kendall's τ and makes similar assumptions about the provided data.

For CONTROL, Kendall's τ and Spearman's ρ coefficients indicated a very weak positive association between CORRECTNESS and DURATION. However, as signified by $p > \alpha'$ (where α' is derived from the adjustment of α as described earlier in this section) for both tests, we fail to reject the null hypothesis $\mathbf{H_0 3}$, since there is insufficient against it because we found no significant positive correlation between CORRECTNESS and DURATION.

For EXPERIMENTAL, both Kendall's τ and Spearman's ρ coefficients indicated a moderate positive association between CORRECTNESS and DURATION. The p-value $p \leq \alpha'$ for both tests indicates that the observed association is statistically significant, i.e. that there is a significant positive correlation between CORRECTNESS and DURATION, so we reject the null hypothesis $\mathbf{H_0 4}$ on the basis of sufficient evidence for the alternative hypothesis $\mathbf{H_a 4}$. Furthermore, the large z-value for Kendall's τ indicates that the observed value for this measure is quite far from the expected value under the null hypothesis, which supports our conclusion that there is a true association between CORRECTNESS and DURATION for this group. Additionally, the values of S yielded by Spearman's ρ test indicate that the observed ranks of the two variables under test are not identical, which provides further evidence that a true association between the variables exists.

6.5.5 Research Question

As described in Section 6.3.8, we defined a research question where we expressed our interest in how accurately participants predicted their correctness ($\mathbf{RQ_{3.1}}$: *How does the exclusion or inclusion of semi-formal MLOps system diagrams affect the accuracy with which participants assess their performance in the context of their perceived task correctness?*). Section 6.3.7 details a survey we required participants to complete after each task, assessing their CONFIDENCE that their answers were correct so that we could determine a SELF-ASSESSMENT score.

We derived a formula that considers both the CORRECTNESS of each participant's answers and the CONFIDENCE they had in the CORRECTNESS of their answers to arrive at an overall SELF-ASSESSMENT score. We give the formula we derived in Equation 6.2:

$$\text{SELF}_{\text{ASSESSMENT}} = \text{SELF}_{\text{CORRECTNESS}} - \frac{5 - \text{SELF}_{\text{CONFIDENCE}}}{5} \tag{6.2}$$

where $\text{SELF}_{\text{CORRECTNESS}}$ represents the participants' average CORRECTNESS as defined in 6.5.3, with 0 indicating entirely incorrect answers and 1 indicating completely correct answers.

$\text{SELF}_{\text{CONFIDENCE}} \in [0, 5] \cap \mathbb{R}$ represents a participant's average CONFIDENCE according to a survey on a five-point Likert scale. We assigned each scale point an equidistant value within this range. A value of 0 indicates that a participant had high CONFIDENCE in the CORRECTNESS of their answers, whereas a value of 5 indicates low CONFIDENCE.

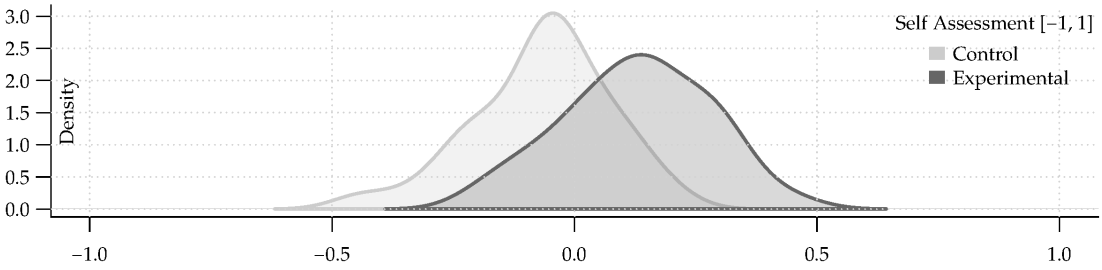

Figure 6.15: Kernel density plot per group of participants' self-assessment.

$\text{SELF}_{\text{ASSESSMENT}} \in [-1, 1] \cap \mathbb{R}$ represents a participant's average SELF-ASSESSMENT score. A value $\text{SELF}_{\text{ASSESSMENT}} < 0$ indicates that a participant overestimated the CORRECTNESS of their answers, $\text{SELF}_{\text{ASSESSMENT}} = 0$ indicates that a participant correctly estimated the CORRECTNESS of their answers and $\text{SELF}_{\text{ASSESSMENT}} > 0$ indicates that a participant underestimated the CORRECTNESS of their answers. Figure 6.15 depicts the participants' overall SELF-ASSESSMENT score per group as a kernel density plot and indicates that CONTROL slightly overestimated their CORRECTNESS on average across all tasks, whereas EXPERIMENTAL slightly underestimated theirs.

6.6 Discussion

6.6.1 Discussion of the Results

Correctness and Duration Overall, our results strongly support the assertion that the provision of semi-formal MLOps system diagrams significantly aids the understanding of MLOps system descriptions when understanding is quantified by CORRECTNESS. Providing practitioners with semi-formal MLOps system diagrams helps them understand MLOps system descriptions and complete tasks based on them. This result aligns with our expectations.

Participants in EXPERIMENTAL did not take significantly longer to complete the tasks than those in CONTROL once they had read and understood the material, despite having more reference material during task completion. A possible explanation for the lack of a significant difference in DURATION is that, when completing the tasks, participants in EXPERIMENTAL may have primarily consulted the UML-based MLOps system diagrams to locate the relevant information and were satisfied that they had found it relatively quickly. Conversely, participants in CONTROL could only search for details in informal textual descriptions and informal graphical system diagrams.

Assuming that DURATION is inversely correlated with understanding, this result shows that providing semi-formal MLOps system diagrams as supplementary material does not hinder understanding. Had it taken participants in EXPERIMENTAL significantly longer to complete the tasks (independent of their CORRECTNESS), this would have indicated that semi-formal MLOps system diagrams can represent a barrier to, or steepen the learning curve for, understanding MLOps system descriptions, at least to some extent.

Consequently, when first attempting to understand a system and complete such tasks, practitioners should not expect to take significantly longer when provided with semi-formal MLOps system diagrams and, in combination with the findings for CORRECTNESS, they can expect better results than if they had not been provided with the semi-formal MLOps system diagrams. This result did not align with our expectations – we expected to see participants taking significantly longer to complete the tasks with semi-formal MLOps system diagrams due to the greater amount of reference material to look through when completing the tasks – but it is, in fact, a positive result since it is a strong argument for providing practitioners with semi-formal MLOps system diagrams.

Correlation Between Correctness and Duration Intuitively, one would expect CORRECTNESS and DURATION to be strongly correlated, and this was our assumption. Contrary to our expectations, there was no significant positive correlation between CORRECTNESS and DURATION within CONTROL. This result suggests that participants in CONTROL believed that they had found the correct answers within a reasonable DURATION[18] and moved on to the next task. Thus, practitioners completing tasks based on informal system descriptions are advised that spending excessive time on such tasks does not necessarily yield better performance in terms of CORRECTNESS.

Conversely and more closely aligned with our expectations, we note a significant positive correlation between CORRECTNESS and DURATION for EXPERIMENTAL. There are multiple possible explanations for this correlation. It could indicate that members of EXPERIMENTAL applied a different methodology when completing the tasks. Given that the only difference between CONTROL and EXPERIMENTAL was the presence of semi-formal MLOps system diagrams, we might assume that members of EXPERIMENTAL mainly relied on the semi-formal MLOps system diagrams as a resource. Alternatively, the positive correlation between CORRECTNESS and DURATION might be explained by the participants first checking the informal textual descriptions and informal graphical system diagrams for answers and then subsequently checking and correcting their answers using the semi-formal MLOps system diagrams (hence leading to an improved CORRECTNESS on average with increased DURATION).

This result greatly strengthens the case for providing practitioners with semi-formal MLOps system diagrams. The additional time spent studying semi-formal MLOps system diagrams has a direct positive correlation with the correct understanding of the modelled systems. Practitioners are advised to utilise model-based architectural documentation whenever possible.

In our study, providing semi-formal MLOps system diagrams appears to be the decisive positive factor, given equal time constraints, with members of EXPERIMENTAL taking a similar DURATION on average to those in CONTROL before being satisfied with their answers, but with EXPERIMENTAL nevertheless performing significantly better in terms of CORRECTNESS.

[18] Any "reasonable DURATION" in this context has, broadly, two potential aspects: (a) the overall time allocated for the experiment session and (b) a possible self-imposed time limit for an individual task, given that participants were conscious that they did not have unlimited time to complete the tasks.

Self-Assessment

In Section 6.5.5, we observed after processing participants' survey responses that the participants whom we did not provide with semi-formal MLOps system diagrams slightly overestimated their performance and conversely. We speculate that the additional semi-formal MLOps system diagrams reduced the CONFIDENCE of those participants whom we provided with them. Those participants possibly felt overwhelmed by the supplementary material or did not feel they had fully understood the semi-formal MLOps system diagrams. Again, this appears to be a counterintuitive result since we would expect these participants to be more confident than their counterparts, especially given that they had performed much better.

We posit that this result is neutral in arguing for or against semi-formal MLOps system diagrams, since practitioners' tendency to be overconfident, underconfident or somewhere in between is more context-dependent, qualitative and subjective than CORRECTNESS and DURATION. We consider this a neutral observation and result, with no positive or negative impact on the hypothesis tests or our conclusions. Still, we nevertheless view it as a factor to bear in mind and deem our associated research question answered within the context of the study.

6.6.2 Threats to Validity

Various kinds of threats to validity must be considered when conducting controlled experiments. We follow the threat classification schemes for experiment validity described by Cook and Campbell [217], and Haynes et al. [218].

Threats to Internal Validity Threats to internal validity encompass variables or circumstances that can potentially generate inaccurate conclusions regarding a cause-and-effect relationship. These elements serve as plausible sources of error that can compromise the capacity to deduce that alterations in the independent variable directly influenced changes in the dependent variable. Threats to internal validity encompass various factors, such as historical events (external influences), maturation (natural changes in subjects over time) or testing effects (how repeated testing impacts outcomes).

Firstly, it should be noted that during the experiment sessions, no disturbances or interference occurred. We provided participants with an introduction and an opportunity to answer questions. No questions arose that affected entire sessions, and we answered participants' questions individually.

Owing to the limited time allocated to each session, the risk of maturation effects was reduced, and we did not observe any. Thanks to the experimental design, each participant contributed only once to the experiment results. Thus, learning between sessions could be ruled out. We consider any learning effect within a session across the three tasks to favour neither CONTROL nor EXPERIMENTAL. Each participant could earn the same number of bonus points for their course, regardless of group. This reward equality precluded instrumental bias, and we prevented selection bias through random assignment of participants to groups.

There may have been potential for cross-contamination between experiment sessions and groups since preventing participants from discussing the experiment with future participants was impossible. Still, since we did not allow participants to take a copy of the experiment sheet with them after their sessions, we did not permit the use of electronic devices during the experiment, and the systems and tasks were quite complex.

We scheduled the sessions either consecutively or several weeks apart; we think it is unlikely that participants gained knowledge unfairly in advance of their sessions. Furthermore, any advantage would likely be roughly even across both groups due to the random group assignment, thus not favouring either group. A ban on electronic devices also prevented participants from consulting external information sources. The only reference material allowed was the printed information document described in Section 6.3.7, to prevent potential effects from participants consulting other sources. Lastly, based on the participant demographics described in Section 6.5.2, we think it is unlikely that either group had an unfair advantage, as neither group dominated the most relevant demographics.

There is a potential risk to internal validity associated with the MLOps system architecture metamodel development and modelling process and, as a consequence, with the models we developed and the resulting diagrams we generated from them and provided to the participants. Although the authors are highly experienced in the applied modelling procedure, the potential for interpretive bias remains. Different researchers may have understood or modelled MLOps systems differently, leading to dissimilar metamodels or system models. However, since the modelling involved developing metamodel constructs that accurately represent observed phenomena in the informal system descriptions, and we successfully used them to model the selected systems, we do not think this risk impacts our study, even if the models created by other researchers may appear different.

The informal diagrams used as source material may also threaten internal validity, as we relied on their accuracy as a basis for the study. Since textual descriptions accompanied the informal diagrams, we could disambiguate any uncertainties in the original informal diagrams. The disambiguation process required a degree of subjectivity, and other authors may have had differing opinions on how to interpret some aspects of the original diagrams. However, since the UML-based diagrams generated from our models accurately reflect the informal descriptions, we anticipate that this phenomenon's impact on our methods and positive findings is minor, if at all.

Threats to External Validity Threats to external validity pertain to constraints on the extent to which study findings can be extended to contexts beyond the specific conditions in which the research was conducted. These threats to external validity raise questions about the broader applicability of the results to different populations, environments or time frames. Notable concerns regarding external validity include selection bias (where the sample may not accurately reflect the larger population) and interaction effects (indicating that the treatment's impact might differ across various groups).

An external validity concern regarding the applicability of our study's findings in practical settings arises from using students rather than non-student professionals. To

address this concern, we implemented measures to educate students on MLOps-related concepts during the experiment. Participants had varying levels of theoretical knowledge in software engineering and distributed systems, as well as programming and industry experience. We are not aware of factors that would preclude a student population of this nature from being representative of a broader population of software developers.

Considering the Stack Overflow industry survey mentioned in Section 6.3.6 in more detail, 69% of respondents to the Stack Overflow survey stated that they are at least partially self-taught and only 45% of developers stated they have a bachelor's degree in computer science or a related field, with only 13% of respondents declaring that they have a master's degree and 2% cent stating that they have a PhD. Thus, most of the 50,000 developers participating in the industry survey, conducted by a well-respected software development portal, were self-taught. Most did not even have a bachelor's degree, so these factors do not appear to be requisite for qualifying as a professional developer, further supporting our claim that students may serve as substitutes for developers in our study.

Consequently, our findings may be extended to professional software developers, at least to some extent. However, to determine whether there are no significant differences compared with the population of professional developers, it would be necessary to replicate similar experiments with practitioners.

A further threat to external validity involves the risk that the tasks we designed may lack relevance to the complexities and nuances of industry practices. This risk introduces a potential limitation that could undermine the generalisability of the study's findings to real-world industry scenarios. We assert that this risk is possible but unlikely, given our experience as an author team and the fact that our perspectives in the study were based on findings from practitioner and scientific literature (see Section 6.3.7).

Threats to Construct Validity Threats to construct validity revolve around uncertainties regarding the degree to which the measurement and operationalisation of variables faithfully reflect the theoretical constructs under investigation. These threats to construct validity pertain to the suitability of the chosen measurement methods. They encompass issues such as insufficient operational definitions, instrumentation that lacks sufficient sensitivity and potential experimenter biases.

As described in Section 6.3.8, we considered CORRECTNESS and DURATION, which are two dependent variables customarily used when measuring understandability. Still, we cannot exclude the possibility that other metrics may be more appropriate for determining understandability. Similarly, more suitable methods may exist for assessing the participants' confidence when addressing our research question.

When participants record their own times, as in this study, it is crucial to ensure that the measurements taken during the experiment accurately reflect the time required to complete the tasks. In this context, various threats to construct validity may arise.

One possibility is that "task completion" was ambiguous or poorly defined, leading to inconsistent timekeeping amongst individual participants across tasks and between participants. We mitigated this risk by instructing participants at the start of each experiment session on precisely when to start and stop recording task time.

Regarding the threat to reliability posed by self-timing, we assessed the reliability of each recorded timestamp during the dataset preparation phase, based on objective exclusion criteria as described in Section 6.5.1. We determined, for each participant, whether their systematic timestamp recording was consistent and realistic. If we determined that a participant's timekeeping was systematically inconsistent or unrealistic, then we excluded this participant's timestamps. For each timestamp, if its value was missing, incomplete or could not be inferred (e.g. due to illegible handwriting), we excluded the time for the entire task. If an individual timestamp yielded an implausible DURATION, we likewise excluded the time for the entire task. By following this methodology, we reduced the risk that unreliable timekeeping would adversely affect our conclusions.

Erroneous measurement accuracy may pose yet another threat to construct validity in the context of self-timing. Owing to participants being instructed to use a single, common clock under our control, with high legibility due to being projected onto a large screen and an accuracy of seconds, we eliminated the risk of inaccurate timekeeping, e.g. due to participants having to start or stop a timer manually, misread the time due to poor legibility, or record inaccurate times due to using different clocks.

A threat to construct validity arises from potential interpretation inconsistencies amongst participants when understanding semi-formal UML diagrams, stemming from diverse interpretations of UML symbols, ambiguity in diagram elements, subjectivity in stereotype usage, varied understandings of relationships, and participants' prior experience with UML. To mitigate this threat, participants in EXPERIMENTAL received supplementary information, including a detailed description of how to interpret the UML-based MLOps system diagrams. Additionally, our pilot test found no ambiguities in the UML diagrams. The statistically significant results favouring the provision of UML diagrams suggest a consistent understanding amongst participants in EXPERIMENTAL, providing reassurance regarding the effectiveness of the mitigation strategies.

Threats to Content Validity Content validity refers to the extent to which the experimental tasks, measurements and materials used in the study accurately and adequately represent the research questions or objectives. In other words, it assesses whether the experiment's content, including variables, tasks and instruments, is relevant and comprehensive enough to measure what the study aims to investigate.

We thoroughly considered content validity in our study design, evaluating the alignment between the experimental tasks, measurements and materials with the hypotheses, research question and objectives. In our meticulous planning, we scrutinised the relevance and comprehensiveness of the variables, tasks and instruments employed in the experiment to ensure they effectively captured and measured the aspects under investigation. Consequently, after a comprehensive review, we could not identify any discernible threats to content validity, as our focus has consistently been on crafting an experimental framework that authentically addresses the problem statement and research objectives outlined in Section 6.1.

Threats to Conclusion Validity Threats to conclusion validity pertain to the extent to which the conclusions derived from a study align with the findings and can be considered well-founded. In essence, considering threats to conclusion validity addresses the accuracy of deducing a causal connection. Threats to conclusion validity may include insufficient statistical power, sampling discrepancies or the excessive extrapolation of results.

Following the experiment sessions, a certain degree of inference and interpretation was necessary, mainly due to the pen-and-paper nature of the experiment, allowing handwritten, free-text responses to some survey and task questions rather than selecting from pre-prepared options. We deliberately made the pen-and-paper format a fundamental part of the experiment design: we did not want to provide pre-prepared answers to choose from since we wanted participants to discover, e.g. component and pipeline names for themselves. Additionally, for specific values, it was more sensible to allow for free text. For instance, for the tasks involving identifying and counting element types, free text made more sense than providing a list of integers to choose from.

Any interpretations or inferences were as follows:

- We interpreted ranges for prior experience in the form of, e.g. "2-3", as "2.5".
- Some participants selected "none" for university education but wrote that they have an unspecified degree in "telecommunications engineering", "business analytics", "biomedical engineering" or "business administration". We recorded these students as having no prior university qualifications since we were only interested in computer science qualifications.
- If a participant left the university education field blank, we assumed they had no relevant university education.
- Some participants entered a non-zero value for the number of years of modelling experience, but did not tick the box signifying prior knowledge of system modelling. These students may have understood these modelling activities as distinct.
- Since participants could enter free text, they often renamed components and pipelines ("elements") when completing the tasks. There were also slight name variations within the source material (both within and between informal textual descriptions and informal graphical system diagrams). Indeed, this is one of the issues that arise with informal MLOps system descriptions and diagrams. We used consistent, standardised naming in our UML-based MLOps system diagrams. Still, when names were vague or inconsistent in participants' answers, we inferred the referenced element when it was evident. We omitted the element from the answer for ambiguous, non-existent or unreadable element names. We also omitted catch-all answers (such as "all model-building components"). We corrected obvious spelling mistakes consistently across both groups, thereby eliminating the risk of bias.

An improvement to enhance the study's construct validity could be to survey participants about their knowledge of MLOps-specific technologies. We used the same list of technologies as in our previous studies for consistency and potential comparison with our previous work.

6.7 Conclusion

6.7.1 Impact and Relevance

We report on a controlled experiment with 63 participants on the understandability of MLOps system architecture descriptions. Our study examines whether providing semi-formal MLOps system diagrams, in addition to informal textual descriptions and informal graphical system diagrams, affects the understanding of the described systems.

Our findings support our assumption that providing supplementary semi-formal MLOps system diagrams significantly aids understanding system descriptions. The respectable score for CORRECTNESS achieved by EXPERIMENTAL, which on average was around double that of CONTROL, whose members scored relatively poorly, provided supporting evidence for this assertion (0.6947 vs 0.3553, respectively). The practical implications are profound: the use of semi-formal MLOps system diagrams significantly enhances task CORRECTNESS, which directly relates to operational efficiency and quality. Practitioners should consider integrating semi-formal diagrams into their documentation and communication practices. The significant difference in task CORRECTNESS in favour of semi-formal MLOps system diagrams suggests that researchers should consider including semi-formal diagrams in their studies. This finding encourages further exploration into how supplementary visual aids, such as semi-formal diagrams, can enhance performance when conducting similar activities to those in this study.

The outcome of our study indicates that, when provided with supplementary semi-formal MLOps system diagrams, it does not take significantly longer to complete tasks of the nature in this study once the provided material has been read and understood. For practitioners, it is thus vital to remember that this additional information, presented as semi-formal MLOps system diagrams, should not lead to the expectation of significant extra time taken to understand a system.

We also found no correlation between task CORRECTNESS and task DURATION when no semi-formal MLOps system diagrams were provided. This result suggests that extra time invested in understanding MLOps system descriptions may not be worthwhile if the system's representation itself is not explicitly designed to enhance understanding. Practitioners should invest their time in developing and using semi-formal MLOps system diagrams for their MLOps systems, as our results indicate a significant increase in CORRECTNESS compared to not using our diagrams, and a measurable positive return in CORRECTNESS for the time spent using these diagrams when understanding MLOps system descriptions. This insight is vital for project planning and resource allocation in real-world MLOps scenarios. The correlation between CORRECTNESS and DURATION in EXPERIMENTAL also highlights an exciting avenue for further research. Investigating the relationships between task CORRECTNESS and task DURATION, with a focus on the impact of semi-formal diagrams, can yield valuable insights into optimising MLOps system descriptions to improve task outcomes.

We noted that members of CONTROL slightly overestimated their CORRECTNESS, whereas their counterparts slightly underestimated their CORRECTNESS. This result serves as a reminder to practitioners not to assume, based on informal textual descriptions

and graphical system diagrams, that they will fully understand the system being described. Conversely, practitioners faced with semi-formal MLOps system diagrams that may appear complex should not necessarily find them daunting. They should be confident that the information they seek is available in the semi-formal MLOps system diagrams. Observing CORRECTNESS estimation behaviour also highlights the importance of self-assessment in practice. Practitioners should be aware of potential bias in their self-assessments and adopt strategies for more accurate self-evaluation. The use of semi-formal diagrams can be considered a best practice for describing MLOps systems in research studies. This approach aligns with the broader scientific principle of improving the replicability and reliability of experiments by providing a standardised visual framework.

We are unaware of previous empirical studies on how best to represent MLOps system architectures to foster understanding. The findings of this study indicate that practitioners should feel confident in adopting a semi-formal MLOps graphical architecture view representation of their MLOps system architectures as a supplementary resource, and that they should expect such diagrams to enhance understanding without the expectation of having to spend significantly more time gaining that understanding.

The results of our controlled experiment, which investigated the impact of using semi-formal MLOps system diagrams, have several implications for education. The significant difference in task CORRECTNESS favouring the inclusion of semi-formal diagrams suggests that incorporating this visual aid in university courses could substantially enhance students' understanding of MLOps system architectures. Students exposed to semi-formal diagrams are more likely to grasp the concepts accurately and apply them effectively.

The observation that participants not provided with semi-formal diagrams tended to overestimate their CORRECTNESS indicates that they may have misconceptions or gaps in their understanding. This observation also highlights the importance of semi-formal diagrams in mitigating the overconfidence effect, which can impede learning. By including these diagrams, instructors can ensure that students' self-assessments better align with their actual knowledge. Whilst there was no significant difference in task DURATION between CONTROL and EXPERIMENTAL, it is essential to recognise the potential trade-off between CORRECTNESS and task DURATION. Students who have access to semi-formal diagrams may invest more time in understanding the material and, in turn, achieve higher CORRECTNESS. Instructors should strike a balance in course design to optimise learning outcomes without overextending course duration or administrative overhead.

Students have diverse learning styles and preferences. Since we observed a significant positive correlation between CORRECTNESS and DURATION for EXPERIMENTAL, educators should emphasise practical, hands-on exercises that allow students to engage with semi-formal diagrams. This hands-on experience is crucial for reinforcing the understanding of MLOps system architectures. MLOps systems encompass both software engineering and ML, making them a suitable topic for interdisciplinary courses. Our findings demonstrate the importance of incorporating visual aids to bridge the gap between these domains, thereby enabling students from diverse backgrounds to collaborate effectively on projects.

Our carefully designed and conducted empirical study builds on our previous work [23], [24] and contributes a solid foundation for further empirical work in the domain of

architectural modelling of MLOps systems. The study also represents a positive initial step towards determining how helpful semi-formal MLOps system diagrams may be in understanding MLOps system architecture. Our results encourage ML practitioners to create and use semi-formal MLOps system diagrams of their MLOps system architectures.

This study helps advance the field of MLOps by promoting the use of semi-formal diagrams as a best practice for improving task CORRECTNESS. These findings stimulate further scientific investigation into the cognitive aspects of CORRECTNESS estimation. Additionally, our findings provide practical guidance for MLOps practitioners, emphasising the potential benefits of semi-formal diagrams in enhancing task CORRECTNESS and the importance of realistic self-assessment.

Finally, our results have the potential to transform how software architecture is taught in university courses. By leveraging semi-formal MLOps system diagrams, educators can improve learning outcomes, address common misconceptions, cater to different learning styles and foster a deeper understanding of complex architectural concepts. Our study also encourages a more holistic, interdisciplinary approach to teaching software architecture, aligning with the industry's evolving demands.

6.7.2 Future Work

Our study examined whether providing semi-formal, UML-based MLOps system diagrams with informal textual descriptions and informal graphical system diagrams enhances the understandability of systems, and its results invite further investigation. There is considerable scope for further empirical evaluations within MLOps system architecture modelling, and this study serves as a suitable starting point.

Given that our UML-based MLOps system diagrams appear to facilitate understanding, an area of potential interest would be to compare different kinds of graphical architecture view representations, e.g. SysML, the C4 model or a novel, custom diagram format to see if understanding in the context of MLOps can be improved further and to discover what factors might help or hinder understanding. Alternatively, textual descriptions could be reconsidered, and a follow-up experiment could compare graphical architecture representations with varying textual architecture design descriptions, as in Jolak et al. [175]. It would also be interesting to compare how different subjects create MLOps system models based on various types of source media (such as informal text and diagrams) to measure consistency and identify features of informal representations that can lead to confusion and ambiguity, and further refine our MLOps system metamodel.

Participants provided with UML-based MLOps system diagrams performed better without taking significantly longer. Still, the recorded times did not consider the effort and time involved in creating our MLOps system models and diagrams. A further study could investigate the trade-off between the time to develop semi-formal MLOps system models and diagrams and the improvement in understanding, to determine whether the additional effort is worthwhile. We would assume that it is worth the effort since EXPERIMENTAL performed significantly better than CONTROL in terms of CORRECTNESS, and CORRECTNESS and DURATION were correlated in our study when we provided semi-formal, UML-based MLOps system diagrams.

As discussed in Section 6.3.6, students may represent professionals in empirical software engineering studies. Nevertheless, it would be interesting to conduct a study with software industry professionals of different backgrounds (e.g. software architecture, software engineering and ML) and with varying levels of experience and perhaps compare them directly with students [169] to gain further knowledge of factors affecting the understanding of MLOps system architecture descriptions.

A future study could focus on semi-formal MLOps system diagrams and utilise eye-tracking as in Sharafi et al. [172], think-aloud protocols as in Heijstek et al. [174], and interviews or surveys to gain insight into which aspects of semi-formal MLOps system diagrams are most helpful, to further refine and improve them. A similar study could investigate how different levels of detail and abstraction influence the understandability of semi-formal MLOps system diagrams.

7 On the Understandability of Machine Learning Practices in Deep Learning and Reinforcement Learning-Based Systems

ML has emerged as a transformative field, utilising various algorithms to enable systems to analyse data and make predictions. DL uses neural networks to tackle complex problems. RL is a method for solving problems by making sequential decisions.

Understanding DL and RL systems solely from their source code is often challenging, especially for inexperienced developers. In a controlled experiment involving 158 participants, we assessed the understandability of DL and RL-based systems and workflows through source code inspection, comparing with semi-formal representations of models.

We hypothesise that DL and RL system diagrams modelling details of DL and RL workflows and practices, such as transfer learning and checkpoints, can enhance the understandability of ML practices in system design comprehension tasks, as assessed through task correctness. Additionally, providing these sources may increase task duration, and we expect a significant correlation between correctness and duration.

In a controlled experiment, the control group, which did not receive semi-formal system diagrams, had an average correctness of 0.7121, whilst the experimental group, which did, had a higher average correctness of 0.7759. On the other hand, participants who received only the system source code showed slightly better performance for correctness (with an average correctness of 0.6808) within the RL-relevant tasks compared to those who also received the semi-formal diagrams (whose average was 0.6612). However, no significant difference was found in the task duration between the two. The control group, for the DL-relevant tasks, took an average of 1571.62 seconds, whereas the experimental group took an average of 1763.85 seconds. For the RL-relevant tasks, the control group took an average of 1883.80 seconds, whilst the experimental group took 1925.46 seconds. Hence, our findings show that providing semi-formal ML system diagrams along with the source code improves correctness for DL-relevant tasks.

7.1 Introduction

In the rapidly evolving field of AI, DL, a domain of ML, has become a highly effective tool for addressing complex problems using neural networks. Similarly, RL has become helpful in solving sequential decision-making tasks and for developing automation [44]. Transfer learning is gaining traction in both DL [45], [46] and RL [47], [48], [49], and is an effective

way to improve learning by transferring knowledge from one domain to another. Transfer learning aims to improve a model's adaptability and applicability in various tasks [1].

In software engineering, ML design models serve as a collection of best practices for addressing common challenges and providing solutions. These models refine the collective expertise and insights of experienced professionals, providing practitioners with widely applicable guidance.

Industry-scale systems often support a wide range of practices and implementations, making it challenging to evaluate whether a given system adheres to best practices and models. In ML, the source code of the ML system is usually the only resource available to understand the ML workflow and the use of best practices and patterns in those workflows [44], [50], [51], [52], making it difficult to comprehend the ML workflows and assess their conformance to best practices and patterns, as the source code of those workflows can be complex and hard to understand. Many techniques exist to implement ML workflows in code, and multiple technologies and library versions further complicate understanding, especially for novice ML software developers.

Today, more and more software developers who are not formally trained in ML have to work on such systems. Conversely, data scientists with limited software engineering education often have to perform software engineering tasks within ML workflows. Design and architectural models, commonly used in other software engineering fields to ease the comprehension of complex source code, remain virtually unused so far for modelling ML workflows.

This chapter is based on **P$_6$** and presents **RC$_6$**. The empirical study aims to answer the following research question in the scope of **RQ$_3$** and **RP$_3$**:

RQ$_{3.2}$ *To what extent does the provision of semi-formal architecture models, in addition to system source code, improve the understandability of DL and RL patterns and practices?*

To address this question, we evaluated participants' understanding of ML practices in the context of DL and RL, experimenting with a total of 158 participants who are trained as software developers but have limited ML experience. We created four groups (see Section 7.4.3). The tasks in the groups in which the participants only receive a system source code repository are called the CONTROL tasks, and the tasks in the four groups in which the participants receive a system source code repository plus help in the form of models are called the EXPERIMENTAL tasks.

We hypothesise that measuring understandability through the dependent variables CORRECTNESS and DURATION would reveal a significantly better understanding of ML practices when models of a given system are provided. More specifically, CORRECTNESS measures the effectiveness and DURATION measures the efficiency of understanding ML practices.

Our results show that participants who were given a system source code repository and semi-formal ML models demonstrated a significantly better understanding than those who only received the system source code repository. Providing models improved understanding of ML practices without compromising overall efficiency in the context of DL.

7.1.1 Structure of this Chapter

We provide an overview of ML practices in Section 7.2. Section 7.3 discusses related work. We describe the experiment planning in Section 7.4. In Section 7.5, we detail the execution of the experiment and the results are presented in Section 7.6. We discuss the results in Section 7.7. In Section 7.8 we describe the threats to validity, and we conclude the chapter in Section 7.9.

7.2 Background

In this section, we provide an overview of the background for this study. We introduce two ML design patterns [1] and the relevant architectural models. These patterns apply in many forms to both DL and RL.

7.2.1 Design Patterns

In ML, design patterns help address the common issues faced when developing and training models. These patterns provide structure and organisation for data handling, model optimisation and effective workflow management. Checkpoints, for example, provide a systematic approach to saving a model's state at regular intervals, enabling the resumption of training with minimal data loss. Transfer learning capitalises on pre-trained models to apply to new tasks, thereby improving performance and reducing training time and costs. Adopting such design patterns leads to better resource allocation, easier model reuse across applications and simplified maintenance.

Checkpoints During ML training, interruptions or failures can result in data loss and inefficient use of computing resources. The challenge revolves around the complex, time-consuming process of training ML models and the risk of interruptions or failures during training. Restarting training from scratch in such cases can be inefficient and result in the loss of valuable data. Dedicated computation and data loss can reduce efficiency, particularly when restarting training is required. Storage and computation needs increase due to the high demands imposed by ML processes [50].

One solution to address these challenges is to use checkpoints [44], [51]. This practice allows storing the complete state of the model regularly during training, facilitating training recovery from a specific point and ensuring the reliability of the workflow. The solution includes key pattern participants and concepts, including creating a checkpoint repository to store and organise checkpoints, defining when and how checkpoints are created, managing checkpoints with assigned names or identifiers and metadata, and utilising checkpoints for purposes such as resuming training, model persistence and fine-tuning. Variations include checkpoints in DL [219], where checkpoints are generated at the end of each training epoch to train the deep neural network and in RL [51], where algorithm checkpoints determine the state of the algorithm used to ensure the continuity of the RL process.

However, this pattern also provides certain drawbacks. Storage and computation overheads are incurred, especially with large models or extended training processes. Managing multiple checkpoints can also become challenging, and there may be increased computational overhead; however, this is typically minimal compared to the benefits gained. Implementing and managing checkpoints can also introduce complexity into the ML pipeline, requiring additional code and considerations about when and how to create them. Additionally, developing and saving checkpoints at regular intervals increases the complexity of ML pipelines, and latency becomes a consideration, with additional pipeline steps impacting overall training time.

Transfer Learning The need for a mechanism to preserve and recover the model state, enabling the recovery of training from specific points, is emphasised by the demand for specialised models for particular cases, such as adapting resource-intensive models to efficiently meet specific objectives [44], [52].

Standard solutions include transfer learning [44], [52], [220], which involves loading a previously trained model that was saved in a model repository. The new model is optimised to address the same problem.

Many forces influence this problem and the resulting solutions. Transfer learning emphasises the efficiency of information. It leverages knowledge from pre-trained models to source functions with sufficient data [44], [52], [221]. This information reveals known properties and reduces reliance on large quantities of job-specific data. Resource efficiency is achieved by reusing the trained model, resulting in faster model convergence and reduced hardware requirements. Transfer learning also supports domain optimisation by adapting to changes in the data distribution.

Transfer learning provides an efficient method for leveraging pre-trained models on large, labelled datasets. This approach eliminates the need for extensive custom datasets. It allows small organisations and specialised applications to develop custom models tailored to specific tasks. They can benefit from collective knowledge by drawing on the wealth of research and insights from others in the field. Additionally, this technique enhances performance for specific tasks [44], [52], [220], [221].

Pattern variants in DL include transfer learning, which entails leveraging a pre-trained model, freezing its weights and integrating non-trainable layers into a new model. In the context of RL, transfer learning enables agents to improve overall performance on related tasks by leveraging knowledge from prior experiences, overcoming challenges posed by data scarcity and time-consuming training in domains such as robotics, gaming, natural language processing and autonomous systems.

7.2.2 ML Scripts and Pipeline Models

The behaviour or workflow of the ML system is usually provided either as an ML script or a pipeline. Both can be modelled as extended (i.e. more detailed) pipeline diagrams. An illustrative script/pipeline diagram is presented in Figure 7.1.

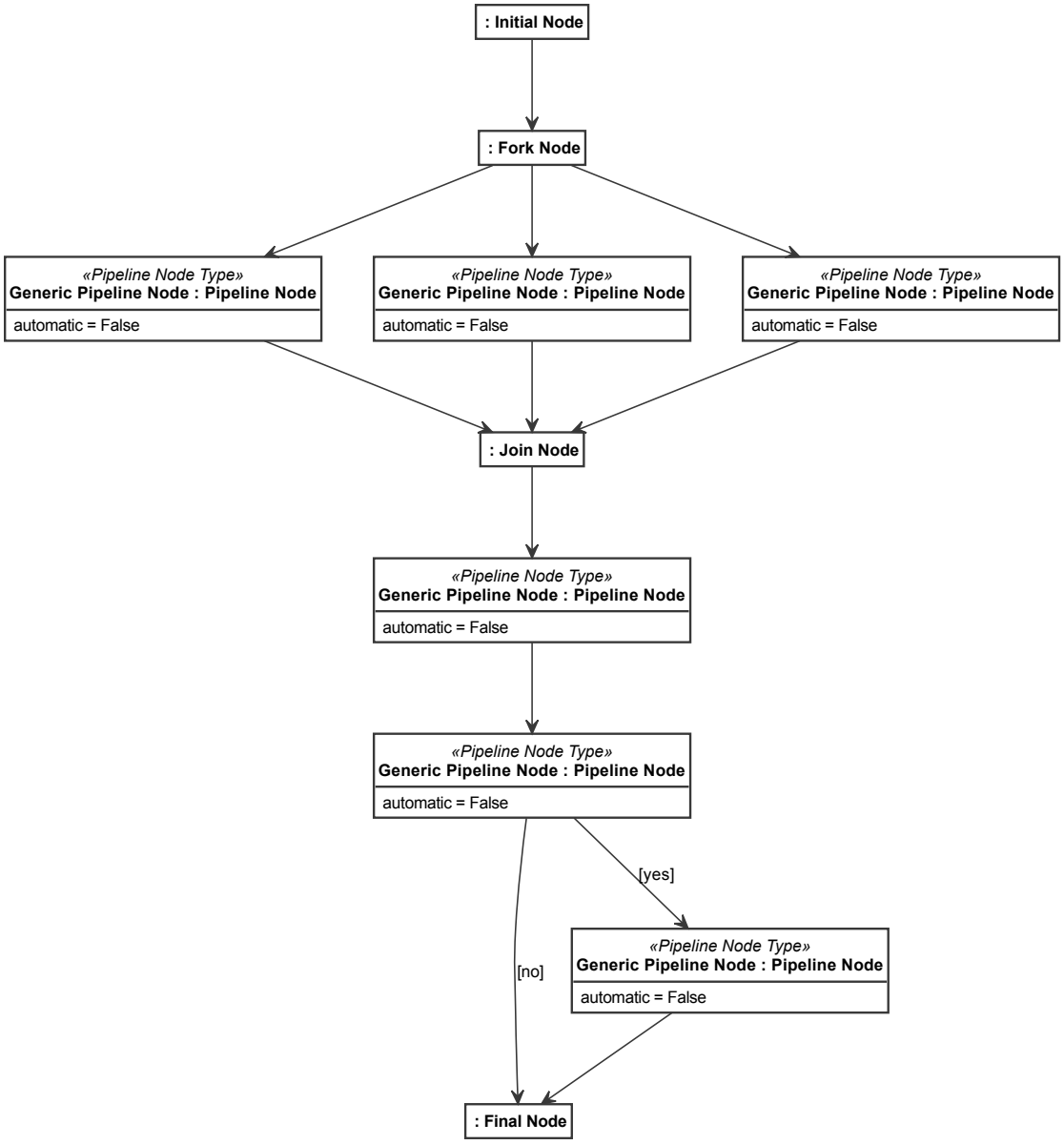

Figure 7.1: Generic pipeline diagram illustrating the elements and relations.

ML scripts are essentially pieces of code that carry out ML tasks step by step. They manage various aspects of the process, including data acquisition, data cleaning, model training and model evaluation to assess its performance. Scripts ensure that all tasks are completed in the correct order, and they automate processes that would otherwise require more time and effort to execute manually. In more advanced systems, such as RL, ML scripts also manage tasks like setting up the learning environment, training agents and saving the model during training, allowing it to be used later.

An ML pipeline can be represented using semi-formal model diagrams. We use and

extend a formal modelling method based on prior work [69] described earlier in this doctoral thesis. All script/pipeline diagrams adhere to a sequential pattern, guiding the flow from an *initial node* to a designated *final node*. These diagrams comprise *pipeline nodes* that symbolise distinct stages within the pipeline. Notable *pipeline node types* encompass steps like *data extraction and analysis*, *data preparation* and *model training* in DL and *environment setup*, *model saving* and *model training* in RL.

Pipeline nodes can have specific properties. Each pipeline node, at a minimum, has an *automatic* property that can assume the values *true* or *false*, indicating whether the step executes automatically. Most steps in a script/pipeline are usually automated. Other *properties* includes *runs in* (indicating the component(s) in which the step operates), *output to* (identifying the component(s) receiving the step's output), *invokes* (identifying the component(s) that are called by the step, such as a task) and *environments* (specifying the execution environment(s) in which the step operates). Please note that the execution environments refer to the cloud or local environments in which the components run, not the RL environments.

Complex pipelines may involve multiple branching paths, denoted as *fork nodes*, that converge at a *join node*. For instance, if there are multiple triggers for a pipeline run, these triggers branch out from a *fork node*, with their paths later merging at a *join node*. Additionally, there are *decision nodes*, which enable a choice of paths and are typically accompanied by annotations that require a decision. Outgoing connections in a pipeline diagram can have labels like *yes* or *no*. For example, if a human needs to decide whether to approve releasing a new model version into production, this decision can be shown with such labels. If a connection is labelled *asynchronous call*, it means the process can proceed without waiting for the result of another step. The primary step continues to run whilst the other one occurs simultaneously.

7.3 Related Work

This section provides an overview of the existing literature on ML best practices and studies that employ comparable methodologies to our research.

Asmaul et al. [52] contribute to the field of transfer learning by presenting a paper that discusses the domain and scope of transfer learning, considering its situational use based on different applications. The paper delves deeply into techniques such as *inductive transfer learning*, *transductive transfer learning* and *unsupervised transfer learning*, covering aspects including sample selection and domain adaptation.

Agrawal et al. [219] make a significant observation regarding the sensitivity of model weights to compression during training. They propose a non-uniform quantisation scheme, an efficient search mechanism to dynamically adjust quantisation configurations and a quantisation-aware delta compression mechanism, all instantiated in DynaQuant – a framework for DL workload checkpoint compression.

Chen et al. [50] introduce *self-ensemble protection* (SEP), a model that leverages the gradients of checkpoints. SEP is effective because it learns from examples ignored during regular training and because the orthogonal nature of checkpoints' cross-model gradients

makes them diverse without requiring the training of multiple models.

Chen et al. [222] propose LC-Checkpoint, a lossy compression scheme for checkpoint constructions, optimising both compression rate and recovery speed. LC-Checkpoint uses quantisation and priority promotion to store crucial information for *stochastic gradient descent* recovery, employing Huffman coding to leverage the non-uniform distribution of gradient scales.

Eisenman et al. [51] introduce Check-N-Run, a scalable checkpointing system designed for training large ML models at Facebook. The system addresses size and bandwidth challenges through differential checkpointing, which tracks and checkpoints the modified part of the model and leverages quantisation techniques to reduce checkpoint size without compromising training accuracy.

In prior work [69], we conducted a controlled experiment with 63 participants to evaluate the understandability of MLOps system architecture descriptions. We compared informal textual and graphical representations with UML-based diagrams. Our results showed that task correctness significantly improves when UML-based diagrams are used. Additionally, we noted that incorporating UML-based diagrams does not significantly extend task duration and thus does not hinder understanding. We also found that there was a significant positive correlation between task correctness and duration, but only when semi-formal diagrams were utilised. Similar to the study described in this chapter, we recruited university students, provided UML-based diagrams as assistance and statistically tested our hypotheses. Notable differences are that our previous study used a between-subject design and focused on MLOps system architecture features rather than DL and RL practices.

Allodi, Cremonini et al. [169] conducted a study involving 73 participants to evaluate the accuracy of security professionals and students with advanced technical education in assessing the severity of software vulnerabilities based on various attributes. In contrast to our focus on comparing different system description methods, they emphasised participants' background knowledge and education. A significant methodological difference is categorising participants into three groups: students enrolled in an MSc in information security program, students enrolled in an MSc in computer science program and security practitioners. Moreover, they specifically recruited students with no professional expertise, distinguishing their approach from ours.

Heijstek et al. [174] conducted a controlled experiment that resembles our investigation. Their primary objective was to assess the efficacy of visual versus textual artefacts in conveying software design decisions to software developers. The study enlisted 47 participants from both industry and academia, who evaluated UML representations as both diagrammatic artefacts and informal textual descriptions. However, there are distinctions between their research and ours. Firstly, all participants in their study assessed both representations, whereas there may be variations in the evaluation process in our study. Secondly, our study explicitly delves into ML practices in the context of DL and RL, thereby setting it apart from the broader scope of Heijstek et al.'s inquiry. These methodological variations emphasise the unique aspects of our study and make a valuable contribution to the broader research landscape in this field.

Labunets et al. [171] conducted a controlled experiment with 29 MSc students to investigate participants' perceptions of visual and textual methods for security risk assessment in terms of effectiveness. Although their study shares similarities with ours in comparing visual and textual representations, it also has notable differences. Firstly, the research does not explicitly examine the distinctions between semi-formal and informal representations, nor does it focus on system understanding, which is a key aspect of our study. Additionally, their experiment does not investigate ML practices, which is our research's primary area of interest. Highlighting these differences shows the specific scope and objectives of our study.

Allodi et al. [170] conducted a controlled experiment with 29 students to explore the challenges participants face when assessing system vulnerabilities as security requirements change. Like our study, they employed a within-subject [82] design, concentrating on variations in system requirements rather than modelling differences. They also formulated hypotheses and tested them using statistical methods, like our approach.

Our work's main contribution is its study of how using UML-based system diagrams to model workflows, including transfer learning and checkpoints, alongside source code, can help people better understand DL and RL systems. Whilst related work has explored practices in ML, such as transfer learning and checkpoints, this study focuses on how these diagrams can assist beginners with limited ML experience. By testing how well participants performed tasks and how long it took them, the research shows that these diagrams improve understanding in certain cases. This work fills a gap in the field by offering practical advice on when and how to use visual aids to make complex ML systems easier to understand, an area that has not been deeply studied before.

7.4 Experiment Planning

Our research aligns with the empirical research principles outlined in the field of software engineering by Jedlitschka et al. [83]. Furthermore, the study design integrates empirical research guidelines in software engineering from sources including Kitchenham et al. [84], Wohlin et al. [85] and Juristo and Moreno [86]. For statistical analysis of the gathered data, the study employs robust statistical methods recommended by Kitchenham et al. [87] for empirical software engineering.

7.4.1 Goals

In this experiment, the goal is to conduct a code and model review of two systems (one DL and one RL) within a 90-minute session and recognise model training patterns and practices used in them. The goal was to assess understanding through code and model reviews, with a specific emphasis on DL and RL scripts. The focus was on reviewing general design aspects and specific recommended model training patterns and practices.

In particular, the checkpoints and transfer learning practices described in Section 7.2 were studied. In both systems, participants reviewed the source code. To assess participants' understanding of the system, each participant received help in the form of

a model of the ML system to be studied in one of the two tasks. In tasks where this help is available, we advised participants to consult the models first, then study the code if more details are required.

7.4.2 System Description

The systems were selected based on their domain and implementation, with a focus on RL in game environments and medical image classification. Both systems utilise advanced techniques, such as PPO and transfer learning, demonstrating different ML methodologies. Source code availability ensures transparency and reproducibility, enabling researchers to efficiently examine and compare results. Moreover, both systems have detailed documentation to facilitate understanding and troubleshooting.

The *RL System using Checkpoints* system[1](Figure 7.2) created in 2023, is a Python-based RL framework that trains agents in the Knights-Archers-Zombies environment using Stable-Baselines3 [223] and SuperSuit for multi-agent training and preprocessing. It uses the PPO algorithm to train agents, with convolutional neural networks and multi-layer perceptrons depending on whether the observations are visual or vector-based. The system applies multiple SuperSuit wrappers, including `black_death_v3`, to handle agent death and ensure a constant number of agents. It trains models for around 81,920 timesteps and evaluates them over 100 games. With around 137 lines of code, it effectively handles agent training and evaluation using vectorised environments and saves trained models. The system is flexible, allowing for easy adjustments to different agent setups and supports multi-agent scenarios through the PettingZoo library. We modified the system to use checkpoints[2].

The *DL System using Transfer Learning* system[3] (Figure 7.3) created in 2020, is a Python-based ML pipeline using TensorFlow, Keras and TensorFlow Hub for transfer learning in both image and text classification tasks. It first trains a model on the colorectal histology dataset [225] (5000 images, 8 classes) using a pre-trained VGG19 [226] model from TensorFlow Hub, with additional classification layers. The system preprocesses the data, creates batches, and fine-tunes the model for 15 epochs, optimising using Adam [227] and categorical cross-entropy[4]. Additionally, it trains a sentiment analysis model using the IMDB reviews dataset, leveraging pre-trained `gnews-swivel-20dim` text embeddings and adds dense layers for binary classification. The total system includes around 168 lines of code for data loading, model construction, training and evaluation. It is optimised for efficient transfer learning by freezing pre-trained layers and focusing on the newly added layers.

---

1 RL source code available at: `https://pettingzoo.farama.org/tutorials/sb3/kaz/`

2 Refer to `Data and Scripts/RL System Source Code/checkpoints_RL.py` in our replication package [224].

3 Transfer learning source code is available at: `https://github.com/GoogleCloudPlatform/ml-design-patterns/blob/master/04_hacking_training_loop/transfer_learning.ipynb`

4 `https://www.tensorflow.org/api_docs/python/tf/keras/losses/CategoricalCrossentropy`

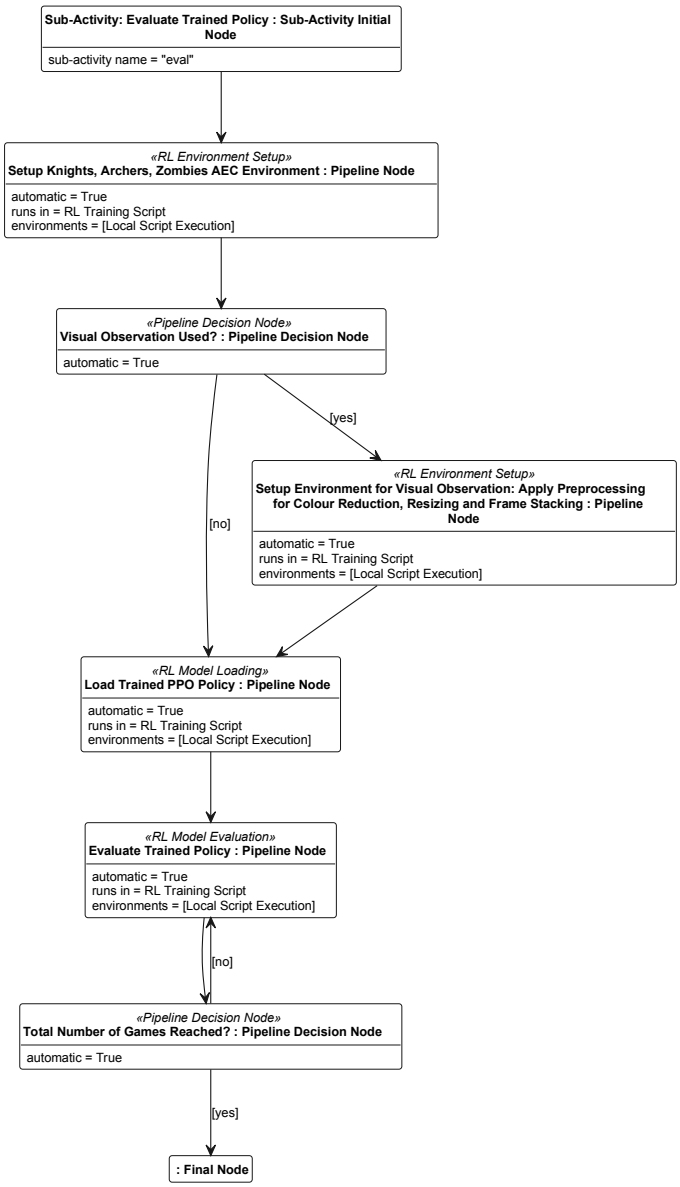

Figure 7.2: Semi-formal model of the RL script pipeline.

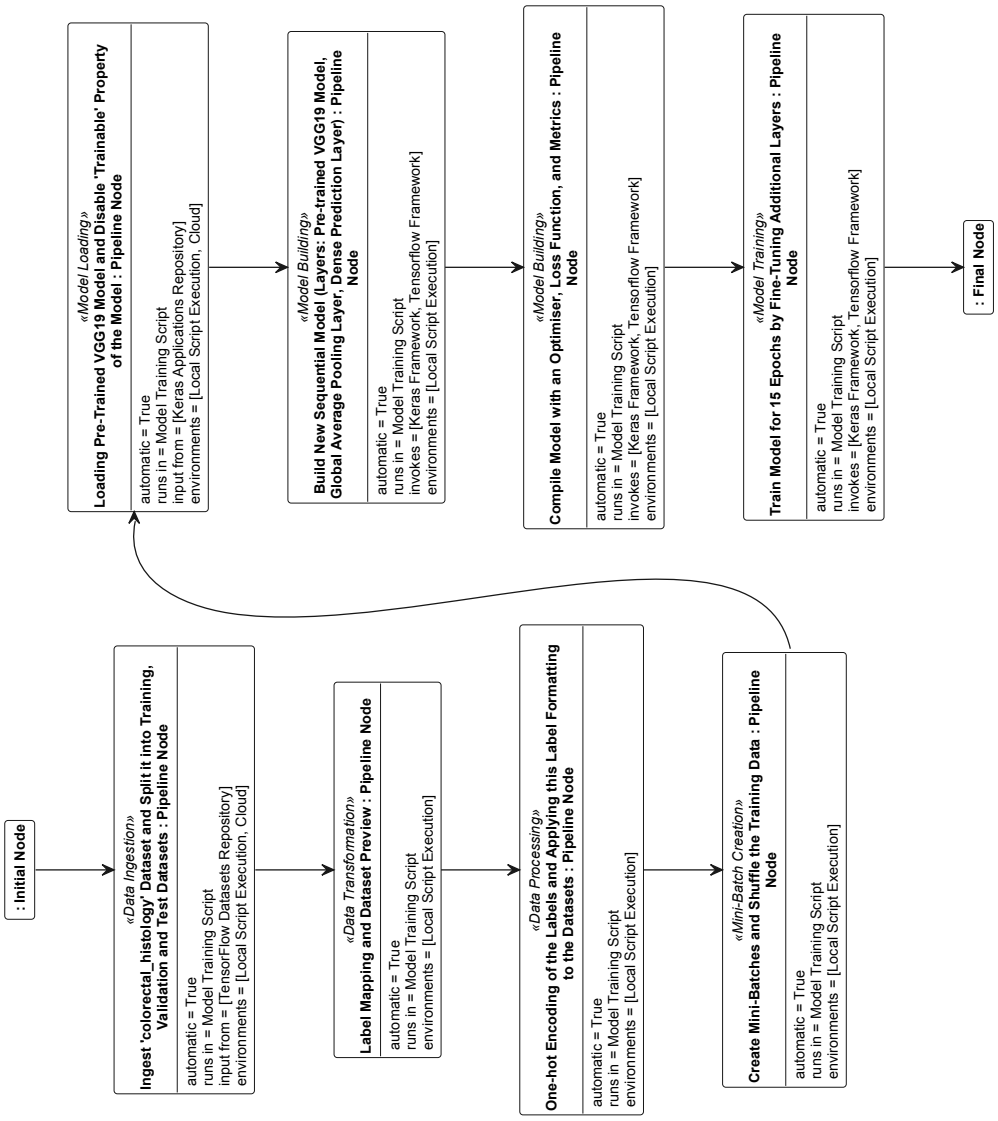

Figure 7.3: Semi-formal model of the DL script pipeline.

7.4.3 Context and Design

In this study, we chose to employ a *within-subject design* based on the insights provided by Charness et al. [82] on the advantages of this experimental approach in specific contexts. In a within-subject design, each participant experiences all conditions, which increases statistical power because each person serves as their own control. This method is advantageous for our study, which aims to identify subtle differences in task performance with and without additional support. Furthermore, within-subject designs align well with theoretical models in which a single individual is exposed to variations in their environment, which aligns with our interest in observing how different support structures

(help versus no help) affect the same participant's performance across two systems.

To minimise potential biases often associated with within-subject designs – such as order effects and demand characteristics – we introduced the reversed-order groups (A2 and B2). This approach addresses the concern that participants may perform differently on the second task due to learning or fatigue from the first task, as highlighted by Charness et al. [82]. By alternating the order of tasks with and without help between groups, we reduce biases and strengthen the experiment's internal validity. This design ensures that any differences in performance are more likely due to the presence or absence of help, rather than the order in which the tasks are completed.

Each student was assigned two tasks, one with help and one without. The two subtasks were called:

- DL System using Transfer Learning (DLS).
- RL System using Checkpoints (RLS).

The groups were composed as follows:

- (A1) DLS: system source code repository and additional help with semi-formal models + RLS: only system source code repository.
- (A2) RLS: only system source code repository + DLS: system source code repository and additional help with semi-formal models; identical to A1, only order reversed.
- (B1) RLS: system source code repository and additional help with semi-formal models + DLS: only system source code repository.
- (B2) DLS: only system source code repository + RLS: system source code repository and additional help with semi-formal models; identical to B1, only order reversed.

7.4.4 Participants

Our study involves advanced bachelor's students enrolled in the Software Engineering 2 (SE2) and Distributed Software Engineering (DSE) courses. Participation in the experiment, which consisted of a hands-on task during the lecture, accounted for 5% of students' overall course points. No further incentives were provided beyond this participation credit. Participants were offered the possibility to engage in the hands-on task and earn course points, whilst opting out of having their contribution included in the experiment. In lectures preceding the hands-on task, topics such as code reviews, design practices and general design patterns were discussed. Participants were also introduced to the experiment through an informational text[5]. This introduction covered background in ML, pertinent DL and RL practices and the modelling techniques employed in the experiment.

Moreover, our participants exhibited diverse knowledge and expertise in software development. A notable reference point is a 2016 survey conducted on the online

[5] Refer to the *information sheet* available in the experiment documents in our replication package [224].

platform Stack Overflow, which involved 50,000 developers[6]. A comparison of our participants' characteristics with those from the survey reveals noteworthy similarities in key demographics. Notably, all participants in our study possessed diverse levels of programming experience, a critical factor for effectively comprehending and reviewing the provided source code materials (see Section 7.6.2 for detailed information).

In our study, we use advanced bachelor's students as proxies for the target population of software developers with limited ML training or data scientists with limited software engineering training. Our study objectives do not revolve around techniques exclusive to a select group of highly trained experts, and as such, we did not specifically target such individuals. In alignment with the findings of Kitchenham et al. [84], utilising students in our study is deemed appropriate, as they represent the next generation of software professionals and share similarities with the population of interest. Their suitability is reinforced by the fact that some participants in our experiment possessed several years of programming and industry experience. Furthermore, corroborating studies by Höst et al. [180], Runeson [181], Svahnberg et al. [182], Salman et al. [183] and Falessi et al. [184] argue that students can serve as valid representatives for professionals in specific empirical software engineering experiments.

7.4.5 Material and Tasks

The experiment is based on a selection of ML practices. The selection includes the practices for *checkpoints* and *transfer learning* summarised in Section 7.2. The selected software practices are related to the subjects taught in the courses SE2 and DSE, including software modelling techniques, software design patterns and software architecture. This study consists of two major experimental material artefacts[7]:

- **(1) Information sheet:** a document explaining the ML practices with an example model that was provided to participants two weeks before the experiment was conducted.

- **(2) Survey form:** four experiment survey forms per group were handed out to participants during the experiment sessions.

All experiment survey forms are structured the same way, consisting of three parts: (1) a participant information questionnaire; (2) two experiment tasks (for one of the four groups A1, A2, B1, B2 explained in Section 7.4.3); (3) an overall experiment questionnaire. Each task is divided into subtasks to test the participants' understanding of DL and RL practices. The participants were instructed to read the code and descriptions in the given repositories for each system before they started to process the tasks. An example from Task RLS.1 follows:

[6] `https://survey.stackoverflow.co/2016`
[7] See our replication package [224].

What is the purpose of using checkpoints during training in this example (multiple options might be correct)?

- ☐ The checkpoints are part of the model evaluation step and are created to record the validation results.
- ☐ Checkpoints are used during training to save the state so far, which is particularly useful if training is interrupted or for later model evaluation and deployment.
- ☐ Multiple agents are trained in parallel in the RL environment. The checkpoints are stored every 1000 training steps using a callback.
- ☐ The checkpoints can be used to compare results across different training stages. While training should usually improve over time, in RL, earlier iterations may perform better than later ones. If this is the case, a checkpoint can be used instead of the model stored after the training.

A task involving the appropriate sequencing of pipeline steps was employed to assess comprehension regarding the sequence of steps in the provided systems. For instance, consider Task DLS.5:

Put the following major steps of the ML script into the correct order in which they are performed. Use numbers like 1, 2, 3 .... Some steps are not part of the script. Add no number for those steps. If a step is performed repeatedly or more than once, only put in the number for the first occurrence and add "*".

_____ Model Building

_____ Model Training

_____ Data Processing

_____ Model Loading

_____ Model Saving

_____ Data Transformation

_____ Data Version Control

_____ Data Ingestion

_____ Data Loading

_____ Model Version Control

_____ Mini-Batch Creation

_____ Model Validation

_____ Model Deployment

_____ Hyperparameter Tuning

In addition to the tasks, a task-based questionnaire was later used to obtain an objective perspective on participants' self-assessment, based on their subjective confidence in the correctness of their answers (see Section 7.6.6). Table 7.1 below outlines the main structure of the questionnaire used in the hands-on tasks for DL and RL.

Table 7.1: Structure of the Questionnaire

Section	Questions/Elements
Demographics	Age (Optional), Gender (Optional), Highest Education Level, Programming Experience, Modelling Experience, Industry Experience, Prior Knowledge in DL, RL and Formal Modelling.
DL Task (DLS)	DLS.1 – Transfer Learning, DLS.2 – Model Layer Configurations, DLS.3 – Epochs Trained, DLS.4 – Component Relationships, DLS.5 – Order of Script Steps, DLS.6 – Automation and Outputs, DLS.7 – Training Practices, DLS.8 – Detailed Relations.
DL Survey	Confidence in answers, ease of understanding system descriptions and diagrams, identifying components and relations, understanding DL/RL practices, component models for large-scale systems.
RL Task (RLS)	RLS.1 – Checkpoints in Training, RLS.2 – Checkpoint Frequency, RLS.3 – Agent Training, RLS.4 – Component Relationships, RLS.5 – Order of RL Script Steps, RLS.6 – Automation and Outputs, RLS.7 – Transfer Learning Relations, RLS.8 – Detailed Relations.
RL Survey	Confidence in answers, ease of understanding system descriptions and diagrams, identifying components and relations, understanding RL practices.
Concluding Survey	Familiarity with textual and graphical descriptions before the study.

7.4.6 Variables and Hypotheses

We formulated the following hypotheses to explore the relationship between experimental conditions and their impact on understanding ML practices. Our controlled experiment measures the following two dependent variables:

- **Correctness** when answering the questions, which includes trying to mark the correct answer and filling in the blanks in the tasks. It is used to measure the effectiveness of understanding ML practices.
- **Duration** being the time it took to complete the experiment tasks (excluding breaks). It is used to measure the efficiency of understanding ML practices.

The two dependent variables, CORRECTNESS and DURATION, may be used to gain insight into the overall understandability of system source code and descriptions [158], [159], [160], [188], [189]. Accordingly, we devised the following null and corresponding alternative hypotheses:

- **$H_0$1** There is no significant difference in CORRECTNESS for EXPERIMENTAL compared to CONTROL.
- **H_a1** There is a significant difference in CORRECTNESS for EXPERIMENTAL compared to CONTROL.
- **$H_0$2** There is no significant difference in DURATION for EXPERIMENTAL compared to CONTROL.
- **H_a2** There is a significant difference in DURATION for EXPERIMENTAL compared to CONTROL.
- **$H_0$3** There is no significant increase in CORRECTNESS as DURATION increases for EXPERIMENTAL compared to CONTROL.
- **H_a3** There is a significant increase in CORRECTNESS as DURATION increases for EXPERIMENTAL compared to CONTROL.

7.5 Experiment Execution

This experiment was executed in three stages: a preparation phase, a pilot test and an execution phase.

7.5.1 Preparation

Two weeks before the experiment, we distributed preparatory materials, including the experiment information sheet (refer to Section 7.4.5), through our e-learning platform Moodle. This document furnished participants with general information about the upcoming experiment and an introduction to ML practices. The document, comprising ML patterns and practices, an illustrative model and a comprehensive description, served as a valuable reference. Participants were permitted to use a printed version of this document during the experiment. The distribution of the experiment information sheet was essential to ensure that all participants were equally well-informed about ML practices in DL and RL, as outlined in Section 7.2.

7.5.2 Pilot Test

After preparing our experimental materials, we conducted preliminary tests with student tutors from our research group. Similar to participants in the experiment execution stage, we provided the tutors with the information sheet two weeks in advance, allowing them to familiarise themselves with the necessary knowledge for the tasks. The tutors participated

in the experiment under predefined conditions, including the option to seek clarifications on the experimental procedure, a 90-minute time frame for responding to experimental questions and no additional support beyond the provided materials.

Following the pilot tests, we gathered feedback from the tutors about their experiences, which suggested that our information sheet was expertly designed, providing clear instructions for tackling the experiment questions. The tutors informed us that we had crafted the information sheet exceptionally well, offering ample guidance for addressing the experiment questions. Given this positive feedback, we maintained our original experiment design.

7.5.3 Execution

The experiment took the form of a closed-book exam, utilising pen and paper, and participants were provided with a printed version of the source code for both systems. Participants were restricted to bringing only the information sheet to aid in completing the experiment tasks. At the commencement of the experiment, each participant was handed a random experiment survey form (refer to Section 7.4.5). Distribution ensured approximately equal numbers of forms of each type (A1, A2, B1, B2 in Section 7.4.3).

Participants were instructed to systematically complete the survey from the first to the last page in the specified order. Additionally, a clock displaying second granularity was projected onto a wall to facilitate timestamp recording. Participants were directed to record start and stop timestamps whilst executing the experiment tasks. Subsequently, the participants' task start and stop timestamps were converted to seconds and summed to determine the total DURATION for all tasks. To safeguard the confidentiality of participant data, an individual not involved in the experiment dissociated personal information (such as name and student number) from the experiment sheets by assigning each participant a unique identification number.

7.6 Analysis

The statistical analysis for the study was conducted using the R [173] programming language version 4.2.2. This analysis comprised several steps, including loading the preprocessed dataset outlined in Section 7.6.1, calculating descriptive statistics for the dependent variables discussed in Section 7.6.4, performing group-by-group comparisons through relevant statistical hypothesis tests as discussed in Section 7.6.5 and generating tables and plots. The reproduction of these results requires the installation of specific R library package dependencies[8].

7.6.1 Data Preparation

The collected raw data[9] from the experiment execution phase (refer to Section 7.5) was prepared as follows: (1) a Microsoft Excel document was exported to a CSV file; (2)

[8] Refer to `Data and Scripts/Scripts/install.r` in our replication package [224].

[9] See `Experiment Documents/Questionnaire Results/experiment-results.csv` in our replication package [224].

the CSV file was imported for further processing; (3) type casting was performed for several data rows; (4) the overall CORRECTNESS for all task CORRECTNESS values was calculated. The dataset is published in our replication package [224], along with all accompanying documents and R scripts.

7.6.2 Participant Demographics

The experiment collects data on participants' age (as depicted in Figure 7.4), gender, course, educational level, programming experience (refer to Figure 7.5), industry experience in software (refer to Figure 7.6) and knowledge of DL and RL practices (refer to Figures 7.7 and 7.8) to capture their background information and experience.

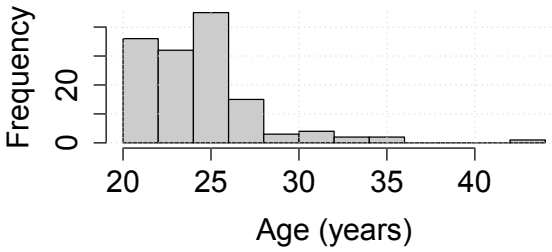

Figure 7.4: Participants' age.

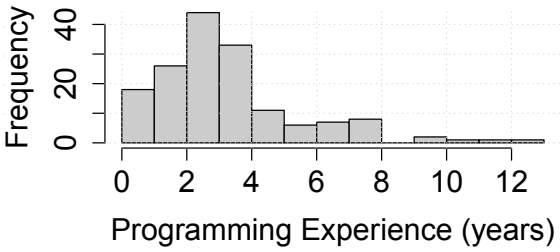

Figure 7.5: Participants' programming experience.

Figure 7.6: Participants' software industry experience.

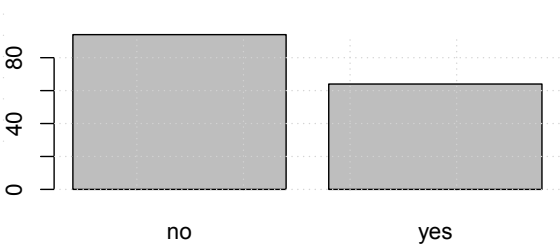

Figure 7.7: Participants' DL experience.

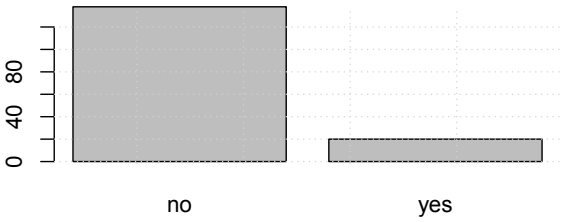

Figure 7.8: Participants' RL experience.

Various tables provide an overview of participants' experience percentages. The data is

compared between two groups: DLS and RLS, each with both experimental and control conditions. Table 7.2 shows participants' experience in the software industry.

Table 7.2: DLS Experimental and Control Group Percentage for Experience in Software Industry

Years	DLS Experimental	DLS Control
0	65.38	58.23
1	10.26	22.78
2	10.26	11.39
3	5.13	2.53
4	5.13	0.00
5	1.28	2.53
6	1.28	1.27
7	0.00	1.27
11	1.28	0.00

The largest group of participants (around 58-65%) reported having no experience. For those with experience, the percentages drop as the years of experience increase. Both the Experimental and Control groups exhibit very similar patterns, indicating that their experience in the software industry is quite comparable.

Table 7.3 shows participants' experience with DL. Most participants (about 58-61%) had no experience, whilst around 39-42% had some. This level of experience applies to both the Experimental and Control groups, meaning their knowledge of DL is quite similar.

Table 7.3: DLS Experimental and Control Group Percentage for Experience in DL

Experience	DLS Experimental	DLS Control
No	60.76	58.23
Yes	39.24	41.77

In Table 7.4, we can see the participants' experience with RL. A large majority (87%) reported no experience, whilst only 13% had some. This level of experience is the same for both the Experimental and Control groups, which suggests that RL is not very common amongst participants.

Table 7.4: DLS Experimental and Control Group Percentage for Experience in RL

Experience	DLS Experimental	DLS Control
No	87.34	87.34
Yes	12.66	12.66

Table 7.5 shows the participants' programming experience. There was a wide range of experience, with many people having two to four years of experience. The largest group (around 25-30%) had three years of programming experience, followed by those with four years (17-24%). The Experimental and Control groups are very similar in this area.

Table 7.5: DLS Experimental and Control Group Percentage for Experience in Programming

Years	DLS Experimental	DLS Control
0	3.80	6.33
1	5.06	7.59
2	16.46	16.46
3	30.38	25.32
4	17.72	24.05
5	6.33	7.59
6	6.33	1.27
7	3.80	5.06
8	7.59	2.53
10	1.27	1.27
11	0.00	1.27
12	0.00	1.27
13	1.27	0.00

Table 7.6 shows participants' experience with modelling. The largest group had two years of experience (26-35%), followed by those with one year (17-21%). Both Experimental and Control groups had similar distributions, meaning not much difference between them.

For RLS, the results are reversed for the Control and Experimental groups because we used a within-subject design. We did not specifically exclude students with limited ML knowledge from the study, as our goal was to include a wide range of experience levels, similar to what is often found in real-world projects. Participants had 0-11 years of industry experience, and their programming experience ranged from 0-13 years. Whilst some were early in their careers, others had more practical knowledge. The study also used questionnaires to gather feedback from students with real-world experience, even if they were not experts.

By focusing on practitioners with limited ML expertise, we aimed to explore how they approach DL and RL technologies. The fact that many participants were not ML specialists is not seen as a weakness, but rather as a realistic reflection of the industry, where many developers must work with ML technologies without formal, in-depth training. We argue that this mix of participants better reflects the current technology landscape, where engineers often use ML in their work without being experts in the field.

Whilst using non-experts might not be ideal for understanding the behaviour of highly specialised ML engineers or data scientists, we assert that doing so accurately reflects

Table 7.6: DLS Experimental and Control Group Percentage for Experience in Modelling

Years	DLS Experimental	DLS Control
0	5.06	13.92
0.33	0.00	1.27
0.5	5.06	2.53
1	17.72	21.52
1.5	3.80	6.33
2	35.44	26.58
2.5	3.80	5.06
3	21.52	12.66
3.5	0.00	1.27
4	5.06	7.59
7	0.00	1.27
8	1.27	0.00
9	1.27	0.00

the real-world experience of many software developers. These developers often face the challenge of integrating ML into their broader software engineering work, even if they lack extensive ML training. We acknowledge that including students with limited ML or DL knowledge may limit the applicability of the results to more specialised groups; however, it still offers valuable insights into how typical developers interact with ML practices.

7.6.3 Normality Assessment

The normal Q-Q plots for DLS (Figure 7.9) for correctness indicate that the data for both Control and Experimental appear to follow a normal distribution. Furthermore, examining the normal Q-Q plots for RLS for correctness (Figure 7.10), it can be inferred

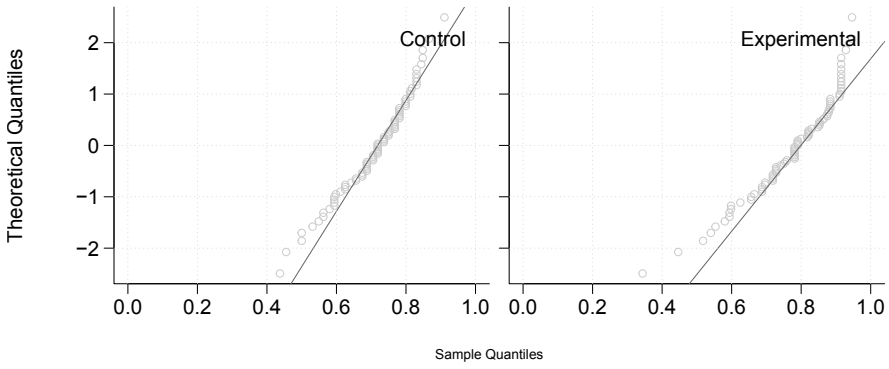

Figure 7.9: Normal Q-Q plot of correctness (DLS).

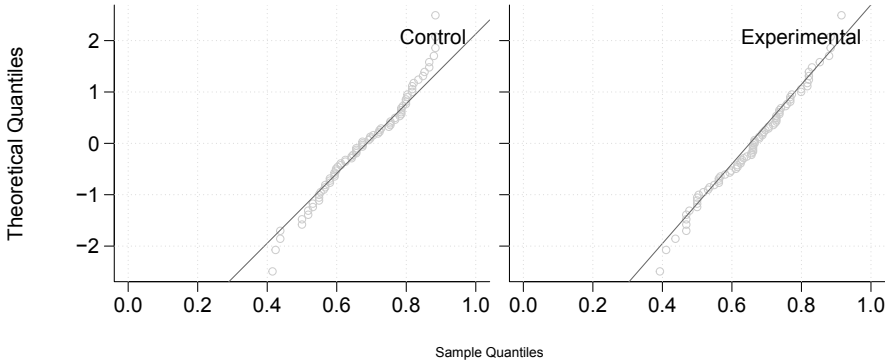

Figure 7.10: Normal Q-Q plot of CORRECTNESS (RLS).

that the data for CONTROL appears normally distributed. However, it is inconclusive regarding the normality of the data for EXPERIMENTAL.

To test for normality, we chose the Shapiro-Wilk [193] normality test, as it is more potent than alternatives (such as Anderson-Darling [194], Lilliefors [195] and Kolmogorov-Smirnov [196]) according to Razali and Yap [197]. Assuming $\alpha = 0.05$, the test for DLS shows that both CONTROL's and EXPERIMENTAL's distributions are significantly different from the normal distribution for CORRECTNESS, with a value $p \leq \alpha$ (see Table 7.7). For RLS, the test indicated that the CONTROL's distribution is significantly different from the normal distribution for CORRECTNESS too, with $p \leq \alpha$, whereas EXPERIMENTAL's distribution does not deviate significantly from the normal distribution, with a value $p > \alpha$ (see Table 7.8).

Table 7.7: Descriptive Statistics per Group of Dependent Variable CORRECTNESS (DLS)

	CONTROL	EXPERIMENTAL
Number of observations	79	79
Mean	0.7121	0.7759
Standard deviation	0.1006	0.1242
Median	0.7188	0.7902
Median absolute deviation	0.0927	0.1324
Minimum	0.4375	0.3438
Maximum	0.9107	0.9464
Skew	-0.6328	-0.9525
Kurtosis	-0.1057	0.7913
Shapiro-Wilk Test p-value	0.0191	0.0002

Visual inspection of the normal Q-Q plots for both groups and both systems for DURATION, visible in Figures 7.11 and 7.12, was insufficient to determine whether each group's data were normally distributed.

Table 7.8: Descriptive Statistics per Group of Dependent Variable Correctness (RLS)

	Control	Experimental
Number of observations	79	79
Mean	0.6808	0.6612
Standard deviation	0.1237	0.1233
Median	0.6741	0.6652
Median absolute deviation	0.1588	0.1324
Minimum	0.4152	0.3929
Maximum	0.8839	0.9152
Skew	-0.1647	-0.1286
Kurtosis	-0.9471	-0.7313
Shapiro-Wilk Test p-value	0.0383	0.3494

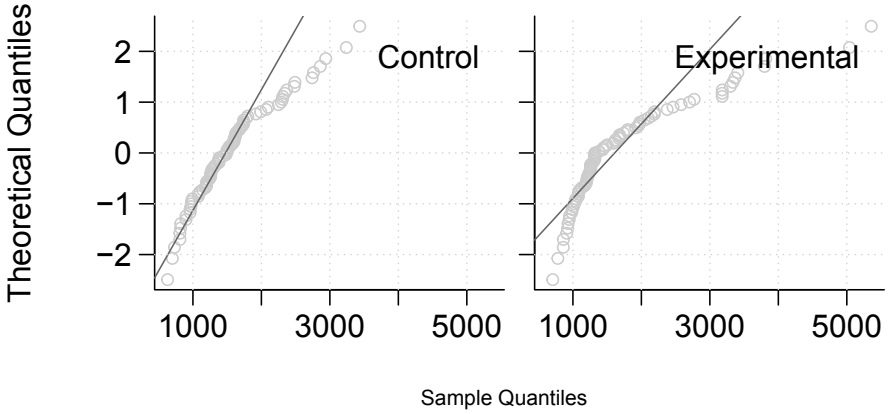

Figure 7.11: Normal Q-Q Plot of duration (DLS).

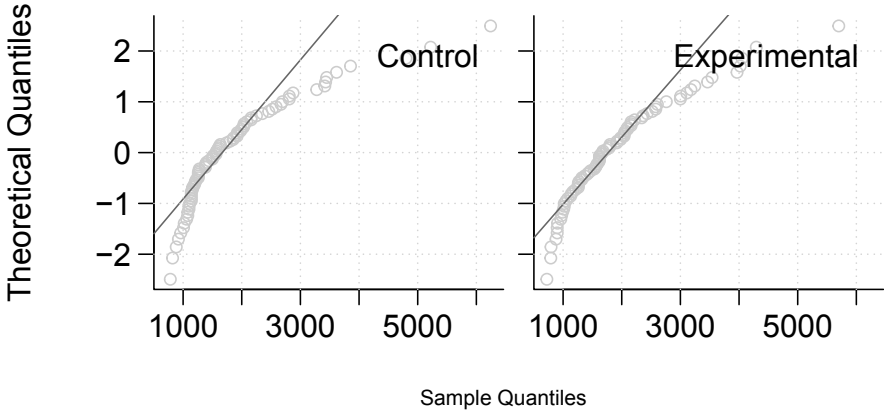

Figure 7.12: Normal Q-Q Plot of duration (RLS).

The Shapiro-Wilk normality test for DLS indicated that both EXPERIMENTAL's and CONTROL's distributions deviate significantly from the normal distribution for DURATION, with $p \leq \alpha$ for each (see Table 7.9). For RLS, the test indicated that, for DURATION, the distributions of the groups also deviate significantly from the normal distribution, with a value $p \leq \alpha$ for both groups (see Table 7.10).

Table 7.9: Descriptive Statistics per Group of Dependent Variable DURATION (DLS)

	CONTROL	EXPERIMENTAL
Number of observations	79	79
Mean	1571.62	1763.85
Standard deviation	607.84	954.04
Median	1489	1320
Median absolute deviation	431.44	518.91
Minimum	630	705
Maximum	3436	5357
Skew	0.9501	1.6488
Kurtosis	0.5306	2.6149
Shapiro-Wilk Test p-value	0.0003	0.0000

Table 7.10: Descriptive Statistics per Group of Dependent Variable DURATION (RLS)

	CONTROL	EXPERIMENTAL
Number of observations	79	79
Mean	1883.80	1925.46
Standard deviation	1019.86	949.80
Median	1560	1670
Median absolute deviation	622.69	726.47
Minimum	782	723
Maximum	6242	5700
Skew	1.9091	1.3771
Kurtosis	4.2426	2.1888
Shapiro-Wilk Test p-value	0.0000	0.0000

7.6.4 Descriptive Statistics

Correctness Table 7.7 and Table 7.8 show the number of corresponding observations, central tendency measures and dispersion measures per group for the dependent variable CORRECTNESS[10] These statistics are illustrated as a kernel density plot in Figure 7.13 and Figure 7.15 and as box plots in Figure 7.14 and Figure 7.16.

[10] CORRECTNESS is defined as a value in $[0, 1] \cap \mathbb{R}$.

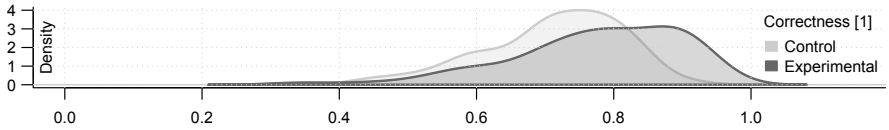

Figure 7.13: Kernel density plot of CORRECTNESS (DLS).

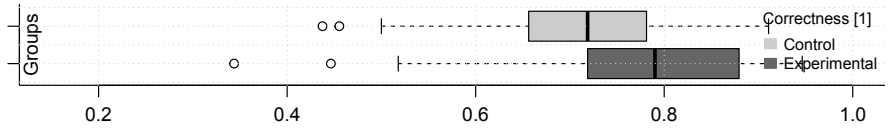

Figure 7.14: Box plots of CORRECTNESS (DLS).

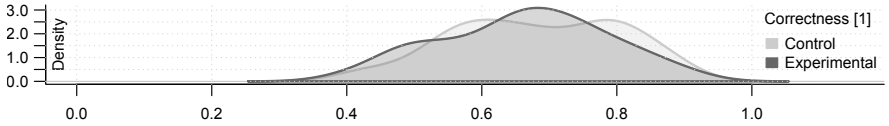

Figure 7.15: Kernel density plot of CORRECTNESS (RLS).

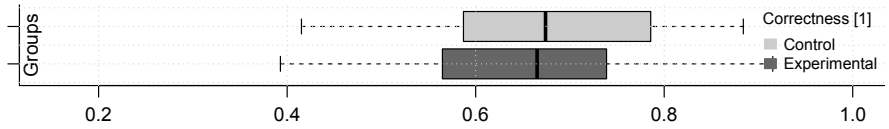

Figure 7.16: Box plots of CORRECTNESS (RLS).

Visual inspection of the DLS CORRECTNESS results (Figure 7.13) and the values in Table 7.7 indicate major differences between CONTROL and EXPERIMENTAL across various statistical measures. The standard and median absolute deviations demonstrate greater variability in EXPERIMENTAL, indicating a wider range of data points. The skewness values indicate that both groups exhibit a degree of asymmetry, with the EXPERIMENTAL having a more pronounced leftward skew. Negative kurtosis in CONTROL suggests a flatter distribution compared to EXPERIMENTAL, which shows a more peaked distribution. The Shapiro-Wilk Test p-values below the conventional significance level ($\alpha = 0.05$) indicate that both groups deviate from normality.

Moreover, for RLS, the statistical measures in Table 7.8 for the CONTROL and EXPERIMENTAL groups are presented. The mean values indicate that CONTROL has a slightly higher average (0.6808) than EXPERIMENTAL (0.6612). Both groups exhibit similar standard deviations (0.1237 for CONTROL, 0.1233 for EXPERIMENTAL), indicating comparable variability. The median values and median absolute deviations are close, suggesting similar central tendencies and dispersion around the median. The minimum and maximum values indicate that the data points span a broader range in EXPERIMENTAL. Skewness values are negative for both groups, indicating a slight leftward asymmetry.

Negative kurtosis values suggest relatively flatter distributions for both groups. The Shapiro-Wilk Test p-values (0.0383 for CONTROL, 0.3494 for EXPERIMENTAL) indicate that CONTROL's data deviates from a normal distribution, whilst EXPERIMENTAL's data is more consistent with normality.

Duration Table 7.9 shows the number of observations, central tendency measures and dispersion measures per group for the dependent variable DURATION[11] Results for DLS shown in Table 7.9 indicate substantial differences in the distribution of the observed values. The mean values show a notable distinction, with CONTROL having a mean of 1571.62 and EXPERIMENTAL having a higher mean of 1763.85. The standard deviation and median absolute deviation are considerably larger for EXPERIMENTAL, indicating greater variability in the data than for CONTROL. The median values indicate that the centre of the data is lower in EXPERIMENTAL (1320) than in CONTROL (1489). The range of values, as reflected by the minimum and maximum, is wider in EXPERIMENTAL. Skewness values reveal that both groups exhibit positive skewness, indicating a tail towards higher values, with EXPERIMENTAL having a more pronounced skew. The kurtosis values suggest that the distributions for both groups are more heavy-tailed than a normal distribution. The Shapiro-Wilk Test p-values are below the conventional significance level ($\alpha = 0.05$) for both groups, indicating a significant difference from normality.

Moreover, for RLS, the data in Table 7.10 indicates differences in the distribution of the observed values. The mean values show that EXPERIMENTAL has a slightly higher mean (1925.46) than CONTROL (1883.80). The standard deviation is lower in EXPERIMENTAL (949.80) compared to CONTROL (1019.86), suggesting less variability in EXPERIMENTAL. The median values indicate that the centre of the data is higher in EXPERIMENTAL (1670) than in CONTROL (1560). The range of values, as reflected by the minimum and maximum, is wider in CONTROL. Skewness values indicate that both groups exhibit positive skew, with the CONTROL group showing a more pronounced skew. The kurtosis values suggest that the distribution for CONTROL is highly heavy-tailed, whilst EXPERIMENTAL is also heavy-tailed but to a lesser extent. The Shapiro-Wilk p-values are similar to those of DLS, with both distributions differing significantly from normality. These statistics are illustrated as a kernel density plot in Figure 7.17 and Figure 7.19 and as box plots in Figure 7.18 and Figure 7.20.

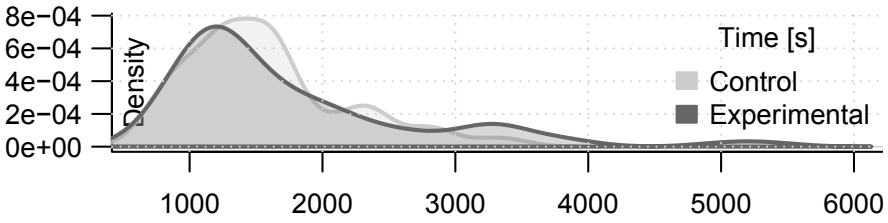

Figure 7.17: Kernel density plot of DURATION (DLS).

[11] DURATION is denoted in seconds.

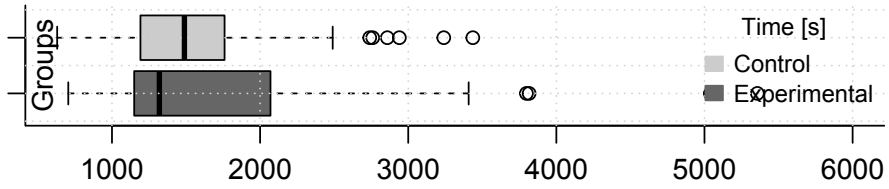

Figure 7.18: Box plots of DURATION (DLS).

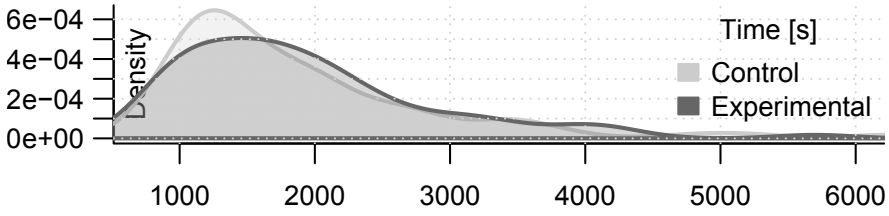

Figure 7.19: Kernel density plot of DURATION (RLS).

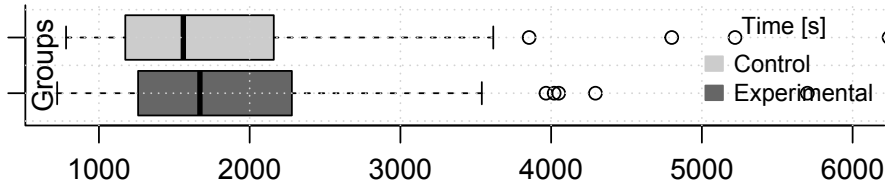

Figure 7.20: Box plots of DURATION (RLS).

7.6.5 Hypothesis Testing

Correctness and Duration Suppose multiple groups are being compared on several dependent variables, as is the case in this study. Then, it is customary to employ the MANOVA [198] statistical test under the condition that specific assumptions are satisfied. This test helps determine whether independent variables impact the dependent variables, individually or in combination. As Section 7.6.4 mentions, the distributions of CONTROL for CORRECTNESS and DURATION significantly differ from the normal distribution. However, the distribution of EXPERIMENTAL does not significantly differ from the normal distribution for CORRECTNESS for RLS. It is important to note that MANOVA requires all distributions to be normally distributed.

As in our prior work [69], we opted to use Cliff's δ [206] as a robust, nonparametric test, as recommended by Kitchenham [87], for scenarios in which distributions differ between populations, or when variances are unequal. Although Cliff's δ was originally designed for measuring ordinal data, it is equally applicable to the quantitative and continuous data used in this study [207], [208]. This test estimates the probability that a randomly selected observation from one group is larger than a randomly selected observation from another group, taking into account the reverse probability [209].

When conducting multiple hypothesis tests using a single method (in this case, Cliff's δ was applied twice), it is necessary to adjust the significance level (α) to mitigate the risk

of type I errors[12]. Several methods can be employed for α adjustment, such as the false discovery rate [210] or the Bonferroni-Dunn [211], [212] correction. The Bonferroni-Dunn correction is the most stringent form of correction and can be calculated using Equation 7.1:

$$\alpha' = \frac{\alpha}{n} \tag{7.1}$$

where n is the number of times a test was applied. In our study, this results in $\alpha' = \frac{0.05}{2} = 0.025$ where $\alpha = 0.05$ and $n = 2$. The results of the one-tailed Cliff's δ test are shown in Table 7.11 for CORRECTNESS and Table 7.12 for DURATION.

Table 7.11: Hypothesis Tests per Group for CORRECTNESS

System: DLS	**CTRL vs EXPR**	**System: RLS**	**CTRL vs EXPR**
Cliff's δ Test		Cliff's δ Test	
Cliff's δ	0.3616	Cliff's δ	-0.0803
s_δ	0.0854	s_δ	0.0922
v_δ	0.0073	v_δ	0.0085
z_δ	4.2336	z_δ	-0.8706
CI (low)	0.1837	CI (low)	-0.2569
CI (high)	0.5166	CI (high)	0.1015
$P(X > Y)$	0.3089	$P(X > Y)$	0.5329
$P(X = Y)$	0.0205	$P(X = Y)$	0.0144
$P(X < Y)$	0.6706	$P(X < Y)$	0.4527
p	0.0000	p	0.3853

For CORRECTNESS, Cliff's δ indicates by $p \leq \alpha'$ that EXPERIMENTAL scored significantly higher than CONTROL for DLS. For DURATION, Cliff's δ yielded $p > \alpha'$, so we cannot conclude that EXPERIMENTAL took significantly longer than CONTROL to complete the experiment.

Based on Cliff's δ, we can reject the null hypothesis $\mathbf{H_0 1}$ for CORRECTNESS for DLS, since there is sufficient statistical evidence to support the alternative hypothesis $\mathbf{H_a 1}$ with $p \leq \alpha'$.

For RLS for CORRECTNESS, we fail to reject the null hypothesis $\mathbf{H_0 1}$, since there is insufficient evidence to support the alternative hypothesis $\mathbf{H_a 1}$ with $p > \alpha'$.

Regarding DURATION, for both DLS and RLS, we again fail to reject the null hypothesis $\mathbf{H_0 2}$, due to insufficient evidence for the alternative hypothesis $\mathbf{H_a 2}$ with $p > \alpha'$.

Correlation Between Correctness and Duration for DLS Upon visually examining the scatter plot in Figure 7.21, which explores potential correlations between the two dependent variables CORRECTNESS and DURATION, no evident linear correlation was

[12] It is important to note that there is no need to adjust the significance level (α) when conducting tests for normality.

Table 7.12: Hypothesis Tests per Group for DURATION

System: DLS	CTRL vs EXPR	System: RLS	CTRL vs EXPR
Cliff's δ Test		Cliff's δ Test	
Cliff's δ	0.0333	Cliff's δ	0.0570
s_δ	0.0929	s_δ	0.0926
v_δ	0.0086	v_δ	0.0086
z_δ	0.3587	z_δ	0.6159
CI (low)	-0.1483	CI (low)	-0.1248
CI (high)	0.2128	CI (high)	0.2351
$P(X > Y)$	0.4799	$P(X > Y)$	0.4679
$P(X = Y)$	0.0069	$P(X = Y)$	0.0072
$P(X < Y)$	0.5132	$P(X < Y)$	0.5249
p	0.7203	p	0.5388

observed for either group. For both CONTROL and EXPERIMENTAL, there was a decrease in CORRECTNESS with respect to time.

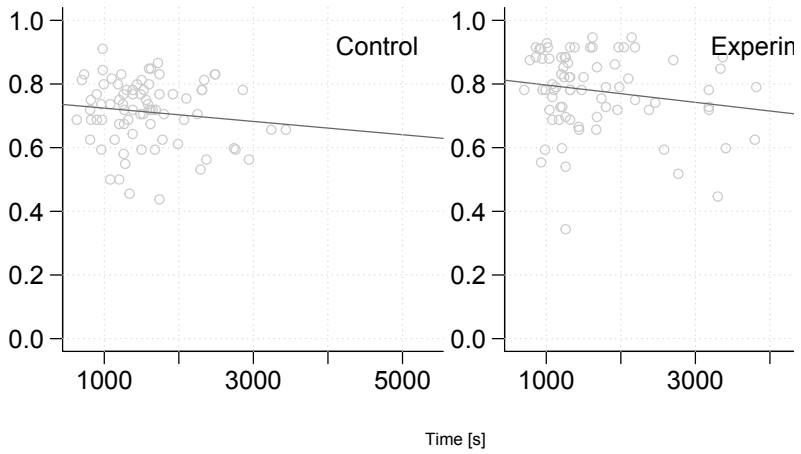

Figure 7.21: Scatter plot per group of the dependent variables CORRECTNESS to DURATION (DLS).

After the visual inspection, we deemed it necessary to perform a correlation test. Spearman's ρ test [216] was selected for this purpose, and the results are shown in Table 7.13.

Control Group Spearman's ρ coefficients indicated a negative association between CORRECTNESS and DURATION. However, given that $p > \alpha'$ (where α' is derived from the earlier adjustment of α), we fail to reject the null hypothesis $\mathbf{H_0 3}$ for DLS, due to insufficient evidence to support the alternative hypothesis $\mathbf{H_a 3}$.

Table 7.13: Correlation per Group of the Dependent Variables CORRECTNESS with DURATION per Group (DLS)

	CONTROL	EXPERIMENTAL
Spearman's ρ	-0.0651	-0.1640
p	0.5688	0.1486
S	87506.3029	95635.5933

Experimental Group Spearman's ρ coefficients also revealed a negative association between CORRECTNESS and DURATION. Despite this, the p-value $p > \alpha'$ suggests that the observed association is not statistically significant, leading us to fail to reject the null hypothesis $\mathbf{H_0 3}$ due to insufficient evidence supporting the alternative hypothesis $\mathbf{H_a 3}$.

Correlation Between Correctness and Duration for RLS

A visual inspection of potential correlations in Figure 7.22 per group did not reveal any significant linear correlation. In the case of CONTROL, there was a minimal decrease in CORRECTNESS relative to time. On the other hand, for EXPERIMENTAL, although there seemed to be a rise in CORRECTNESS with DURATION, the data points were widely dispersed from the indicated reference line, making it inappropriate to assume a linear correlation.

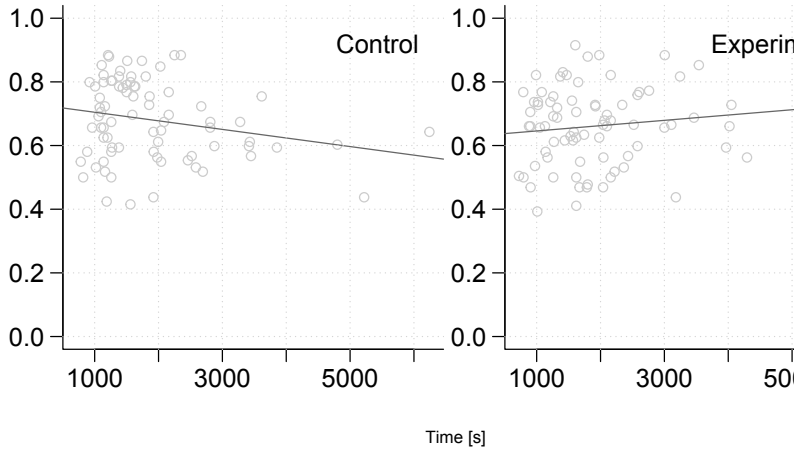

Figure 7.22: Scatter plot per group of the dependent variables CORRECTNESS to DURATION (RLS).

Control Group Spearman's ρ coefficients, shown in Table 7.14, yielded similar results to DLS, indicating a negative association between CORRECTNESS and DURATION. However, given that $p > \alpha'$, we also fail to reject the null hypothesis $\mathbf{H_0 3}$, due to insufficient evidence to support the alternative hypothesis $\mathbf{H_a 3}$.

Experimental Group Spearman's ρ coefficients showed a very weak positive association between CORRECTNESS and DURATION. As with DLS, the p-value $p > \alpha'$ suggests

Table 7.14: Correlation per Group of the Dependent Variables CORRECTNESS with DURATION per Group (RLS)

	CONTROL	EXPERIMENTAL
Spearman's ρ	-0.1185	0.0811
p	0.2982	0.4773
S	91897.0184	75495.2409

that we cannot reject the null hypothesis $\mathbf{H_0 3}$, due to insufficient evidence for the alternative hypothesis $\mathbf{H_a 3}$.

7.6.6 Observation

Following the details provided in Section 7.4.5, participants were instructed to fill out a survey after each task to indicate their CONFIDENCE in the accuracy of their responses. A SELF-ASSESSMENT score was calculated using a formula that considers both the CORRECTNESS of participants' answers and their level of CONFIDENCE in the CORRECTNESS. The formula for calculating the SELF-ASSESSMENT score is presented in Equation 7.2:

$$\text{SELF}_{\text{ASSESSMENT}} = \text{SELF}_{\text{CORRECTNESS}} - \frac{5 - \text{SELF}_{\text{CONFIDENCE}}}{5} \tag{7.2}$$

where $\text{SELF}_{\text{CORRECTNESS}}$ represents the participants' average CORRECTNESS as defined in 7.6.4, with 0 indicating entirely incorrect answers and 1 indicating completely correct answers.

The variable $\text{SELF}_{\text{CONFIDENCE}} \in [0, 5] \cap \mathbb{R}$ represents the average CONFIDENCE of participants based on a survey that utilised a five-point Likert scale. Each point on the scale is assigned equidistant values within the range of 0 to 5. A value of 0 indicates high CONFIDENCE in the CORRECTNESS of the answers, whilst a value of 5 indicates low CONFIDENCE.

The variable $\text{SELF}_{\text{ASSESSMENT}} \in [-1, 1] \cap \mathbb{R}$ represents the average SELF-ASSESSMENT score of participants. A value less than 0 ($\text{SELF}_{\text{ASSESSMENT}} < 0$) indicates that participants tend to overestimate the CORRECTNESS of their answers. A value of 0 ($\text{SELF}_{\text{ASSESSMENT}} = 0$) indicates that participants accurately estimate the CORRECTNESS of their answers. A value greater than 0 ($\text{SELF}_{\text{ASSESSMENT}} > 0$) indicates that participants tend to underestimate the CORRECTNESS of their answers.

Figure 7.23 illustrates the kernel density plot of participants' overall SELF-ASSESSMENT score per group. The plot shows that, on average, participants in both CONTROL and EXPERIMENTAL groups underestimated their CORRECTNESS across all tasks. Similarly, Figure 7.24 indicates that both CONTROL and EXPERIMENTAL groups underestimated their CORRECTNESS.

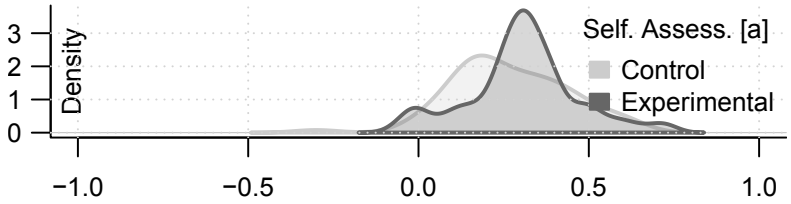

Figure 7.23: Kernel density plot per group of participants' SELF-ASSESSMENT (DLS).

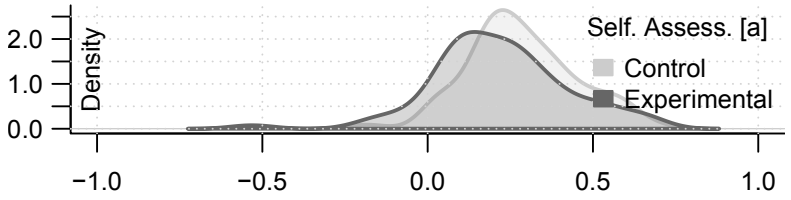

Figure 7.24: Kernel density plot per group of participants' SELF-ASSESSMENT (RLS).

7.7 Discussion

7.7.1 Interpretation of the Results

Correctness and Duration

In the DLS study, our investigation of $\mathbf{H_0 1}$ revealed a significant finding. There was a significant difference in task CORRECTNESS, indicating that providing semi-formal diagrams alongside source code improves performance. However, when we tested $\mathbf{H_0 2}$, we found no significant difference in task DURATION between CONTROL and EXPERIMENTAL. Regarding $\mathbf{H_0 3}$, we observed that for both CONTROL and EXPERIMENTAL, there was no significant correlation between CORRECTNESS and DURATION.

In contrast, during the RLS study, our examination of $\mathbf{H_0 1}$ showed that CONTROL performed similarly to, although slightly better than, EXPERIMENTAL in task CORRECTNESS. This result can be attributed to several factors, including the relatively small size of the system's source code and the added complexity introduced by multiple diagrams. Participants in the EXPERIMENTAL group may have struggled to fully understand some of the details in these diagrams, leading to minor misunderstandings. Meanwhile, those in the CONTROL group could more easily grasp the underlying practices because they had to deal with only a modest amount of source code.

Additionally, we noticed that modelling recurring elements as a sub-activity within another model might have introduced unnecessary complexity. To address this, we included a third sub-model; however, it became clear that these sub-models might have introduced structural complexities similar to those in the source code itself. This observation suggests that models must reach a certain level of abstraction to be truly useful, although determining the exact threshold for this abstraction remains an area for future research.

Similarly, when testing $\mathbf{H_0 2}$, we found no significant difference in task DURATION between CONTROL and EXPERIMENTAL for either system. Moreover, in evaluating task DURATION, it was apparent that participants in the EXPERIMENTAL group took slightly more time to answer the questions. This result is expected given the additional cognitive load associated with analysing both the source code and the accompanying diagrams.

We must recognise that the inherent complexity of the DL system source code poses unique challenges. It requires a careful effort to identify and understand the various practices embedded within them. When we contrast the modelling approaches of RLS and DLS systems, an interesting observation emerges. The RLS model delved into finer details, offering a comprehensive depiction of the workflow, whilst the DLS model embraced abstraction, providing a broader overview. This difference suggests that the value of models may lie in their ability to offer a level of abstraction distinct from the source code. It leads us to consider that, if the source code is available, models might be most beneficial when operating at a higher level of abstraction. We employed different modelling strategies in developing these two systems, each serving a unique purpose.

Correlation Between CORRECTNESS and DURATION

When testing $\mathbf{H_0 3}$, we discovered that within CONTROL, no significant positive correlation between CORRECTNESS and DURATION was discerned; conversely, within EXPERIMENTAL, a weakly positive, albeit insignificant, correlation between CORRECTNESS and DURATION was detected for RLS.

Our initial impressions and intuition about DLS led us to expect a positive correlation between CORRECTNESS and DURATION. However, contrary to our expectations, in the case of CONTROL, there was no positive correlation between CORRECTNESS and DURATION, suggesting that participants in CONTROL were satisfied with their answers within a reasonable time and continued accordingly.

On the other hand, in the case of RLS, as shown in Figure 7.22, it is clear that EXPERIMENTAL shows some improvement with more time in CORRECTNESS in contrast to CONTROL. Consistent with our initial hypothesis, our results for EXPERIMENTAL in both CORRECTNESS and DURATION show a positive correlation, although not significant. It is possible that participants in EXPERIMENTAL adopted a different strategy for answering the questions. Since the only difference between CONTROL and EXPERIMENTAL was the inclusion of semi-formal diagrams, it is plausible that participants heavily relied on these resources in EXPERIMENTAL. Alternatively, participants may have initially consulted the source code for answers and then used the semi-formal diagrams to review and edit their answers, leading to a weak positive relationship between CORRECTNESS and DURATION.

Therefore, practitioners working on source code should be aware that spending more time on such tasks does not necessarily lead to better performance in terms of CORRECTNESS. Participants in CONTROL spent a similar length of time on average as participants in EXPERIMENTAL before being satisfied with the answer. This behaviour suggests that other factors may affect the performance.

7.7.2 Addressing the Research Question

We set out to answer a single research question:

RQ$_{3.2}$ *To what extent does the provision of semi-formal architecture models, in addition to system source code, improve the understandability of DL and RL patterns and practices?*

In DLS, we found that providing participants with models and their source code significantly improved their CORRECTNESS in answering questions, indicating that they better understood ML practices. However, the extra help did not make them complete the tasks significantly faster. In cases where the source code was simple, like RLS, just having the source code was enough for participants to understand the source material well. The extra diagrams may have actually caused some confusion.

Overall, the study suggests that semi-formal models can enhance understanding, but their usefulness depends on the task's complexity and the clarity of the models' design. For straightforward tasks, source code alone might suffice, but for more complex systems, models can make a positive difference. We also found that in RLS, participants who used the models took longer but were slightly more correct in their answers, suggesting that the models helped them think more deeply. So, whilst models can be useful, they need not be complex to be helpful for developers new to ML.

7.7.3 Self-Assessment

In Section 7.6.6, our examination of participants' survey responses in the context of DLS yielded a curious finding: those who did not receive semi-formal ML system diagrams estimated their performance slightly more accurately than those with the additional materials. Conversely, in RLS, participants who did not receive semi-formal ML system diagrams tended to underestimate their performance slightly more than those who did. The provision of additional semi-formal ML system diagrams may have contributed to a reduction in confidence levels in certain contexts, like DLS. These participants may be overwhelmed by the information or have doubts about their complete understanding of semi-formal ML system diagrams.

This result deviated from our initial expectations, as we expected higher CONFIDENCE amongst those participants who received extra help. However, this finding does not strongly support or oppose the use of semi-formal ML system diagrams. Practitioners' tendencies towards overconfidence, underconfidence or a position somewhere in between are subjective and context-dependent; they cannot be deduced solely from the relationship between CORRECTNESS and DURATION. Therefore, this observation remains neutral and does not affect our conclusions. However, we acknowledge the importance of careful consideration throughout our analysis.

7.8 Threats to Validity

Threats to Internal Validity
The experiment proceeded without any problems that could affect the experiment execution. Participants received clear instructions and an opportunity to ask questions. There were no major concerns that affected entire sessions, and any questions were answered individually.

The limited time for each session helped reduce the chance of any changes over time, and no such effects were observed. Each participant took part in only one session, thereby preventing any learning that might have occurred between sessions. Any learning that happened within a session did not give an advantage to either CONTROL or EXPERIMENTAL. All participants had an equal chance to earn points based on their performance, regardless of the group they were in, thereby avoiding any scoring bias. The random assignment of participants to groups also helped prevent selection bias.

Although it was not possible to completely prevent participants from discussing the experiment with others, steps were taken to minimise the likelihood of this happening between sessions. Participants were not allowed to take any materials with them or use electronic devices during the experiment. The complexity of systems and tasks, including the differences between activities, reduced participants' opportunities to gain an advantage during the session. Random group allocation aimed to ensure an even distribution of advantages and disadvantages. The ban on electronic devices also prevented participants from accessing outside information. As described in Section 7.5.3, the only authorised materials were the printed information sheets described in Section 7.4.5, which, together with restricting access to the source code, prevented participants from using other external sources.

Threats to External Validity
One issue that may affect the external validity of our study is the sample size, which may not be sufficient to yield statistically significant results. To address this concern, we employ robust statistical methods tailored to the size of our sample and ensure that the analysis was carried out appropriately and accurately, given the available data.

Another consideration arises from the use of students rather than non-student professionals in our study. This choice of participant raises questions about the generalisability of our findings to practical settings. To alleviate this problem, we took measures to familiarise students with the ML-related concepts used in the experiment. All participants have varying levels of theoretical background in software engineering and distributed system programming, as well as industry experience. According to the Stack Overflow industry survey (refer to Section 7.4.4), 69% of respondents were self-taught, 43% held a bachelor's degree in computer science or a related field, 19% had a master's degree, and 2% had a PhD. These statistics indicate that a significant portion of the 50,000 developers surveyed, even on a widely respected software development platform, were self-taught and lacked formal degrees. The statistics also suggest that a degree may not

be required to qualify as a professional developer.

Given this context, we assert that students can reasonably serve as substitutes for developers in our study. Consequently, we propose that our findings may apply to professional software developers, at least to some extent. However, it would be prudent to replicate similar experiments with practitioners to confirm that there are no significant differences compared to the professional developer population.

Whilst developers typically use IDEs in real-life scenarios, our experiment was designed to test how well people understood the code when diagrams were included, rather than how quickly they could navigate it. Using printed code helped us see how the diagrams affected understanding without other factors interfering. We acknowledge that using IDEs may affect task completion time, but our controlled setup still provides valuable insights.

Threats to Construct Validity

Threats to construct validity occur when there is uncertainty about whether the methods used to measure and define variables truly represent the concepts being studied. These challenges can include doubts about the accuracy of measurement methods, unclear definitions, instruments that are not sensitive enough or bias introduced by the researcher. In Section 7.4.6, we introduced CORRECTNESS and DURATION as dependent variables to assess understandability, but we acknowledge that other metrics might be more appropriate. Additionally, there may be more effective ways to measure participants' CONFIDENCE in answering our research question.

It is essential to ensure that the times recorded by participants accurately reflect the time they spent completing tasks. Threats to construct validity here could include unclear task definitions, which may lead to inconsistent timekeeping. We attempted to mitigate this risk by providing participants with clear instructions and guidance.

To ensure reliable self-timing, we carefully checked the reliability of all recorded timestamps when preparing the dataset, removing any inconsistent or unrealistic entries. We also addressed any missing or implausible time values to reduce the chance of unreliable time data affecting our results. Participants were required to use a centrally controlled clock with high readability and accuracy to avoid timing errors, such as manual manipulation or misreading, and to ensure consistency across all sessions.

We carefully created diagrams to clearly and structurally show the workflows of both systems (DLS and RLS). However, because RLS was more complex, we needed multiple diagrams to cover all the details. Providing multiple diagrams might have made the details harder to understand for RLS than for DLS, leading to a larger gap between the diagrams and the code, which may be one reason the results for RLS differed.

To ensure the diagrams were equally helpful in both systems, we designed the questions to the same level of detail, making them equally informative. We also reviewed the diagrams to ensure they were accurate and captured the main aspects of each system without adding unnecessary complexity. Still, we recognise that the larger gap between the RLS diagrams and the code may have affected participants' understanding of the system, and this is something future studies could investigate further.

Threats to Content Validity
We consider the content free of any threat to its integrity because the topics under test are related to the university courses of all participants. Regardless of their group assignment, the provided information materials also provided sufficient background knowledge to enable participants to effectively take part in the study. Any inconsistencies or ambiguities in the experimental material would have affected both groups equally.

7.9 Conclusion

Our investigation into DLS and RLS offers fascinating insights into the effectiveness of incorporating semi-formal ML system diagrams alongside source code. Our analysis in DLS shows significant differences in task CORRECTNESS, indicating that semi-formal ML system diagrams, together with the source code, contribute to better results without a significant increase in DURATION.

Regarding RLS, our investigation revealed that CONTROL performed comparably or slightly better than EXPERIMENTAL for CORRECTNESS (although not significantly), contrary to expectations. We found no significant difference in DURATION between CONTROL and EXPERIMENTAL. We suspect that EXPERIMENTAL participants faced challenges in understanding the complex details in the diagrams and the complexity of multiple diagrams. In contrast, participants in CONTROL were required to handle only reasonably sized source code and therefore, displayed a better understanding. Additionally, we found a weak positive (but insignificant) correlation between CORRECTNESS and DURATION within EXPERIMENTAL, suggesting a distinct approach to answering questions amongst participants.

Examining the participants' survey responses yielded some interesting findings. In DLS, participants who did not receive semi-formal ML system diagrams tended to estimate their performance slightly more accurately. In contrast, in RLS, participants who did not receive these diagrams underestimated their performance more than their counterparts. This outcome highlights the impact of additional materials on participants' CONFIDENCE.

In conclusion, whilst providing semi-formal ML system diagrams appears beneficial in specific contexts, their effectiveness varies across different learning environments. The decision to incorporate these diagrams should be made with careful consideration of practitioners' specific objectives.

Part V

Assessing Conformance to Quality Characteristics, Best Practices and Patterns in MLOps and RL-Enabled Systems

8 A Model-Driven, Metrics-Based Approach to Assessing Support for Quality Aspects in MLOps System Architectures

In MLOps, automation is a fundamental pillar that streamlines the deployment of ML models and serves as an architectural quality aspect. Support for automation is especially relevant when dealing with ML deployments characterised by the continuous delivery of ML models. Taking automation in MLOps systems as an example, we present novel metrics that offer reliable insights into support for this vital quality attribute, validated by ordinal regression analysis. Our method introduces novel, technology-agnostic metrics aligned with typical ADDs for MLOps automation. We systematically demonstrate the feasibility of our approach in evaluating automation-related ADDs and decision options. Our approach can itself be automated within CI/CD pipelines. It can also be modified and extended to evaluate any relevant architectural quality aspects, thereby assisting in enhancing compliance with non-functional requirements and streamlining development, quality assurance and release cycles.

8.1 Introduction

ML systems pose unique challenges compared to traditional software architectures due to the intricate interplay of software development, data science, data engineering, ML models-as-a-service and ML-related end service engineering in the form of MLOps, as well as parallels with DevOps. Noteworthy challenges in ML application software structure include complex dependencies between software modules [53], technical debt and anti-patterns [38], and design challenges in ML model engineering, such as model selection and reuse [54], [55]. Model provisioning and production runtime monitoring of ML service performance, often in a distributed environment, further exacerbate complexity and emphasise the intricacies in organising and decomposing ML systems. Paleyes et al.'s [168] survey of case studies identifies practical challenges in ML system architecture, encompassing tools, services and architectures within the deployment structure, as well as potential disparities between ML, data science and software engineering practices. This complexity has led to the emergence of AutoML [162], [163] and MLOps [4], [5], [228] as sub-disciplines within DevOps [21], [229] and CD [22].

Studies on open-source systems [164], the scientific literature [165] and practitioner insights [166] highlight that adopting DevOps and CD results in a complex deployment

and delivery architecture. This complexity extends beyond the intended system's software architecture. It becomes even more pronounced in MLOps systems, where additional aspects such as ML model management and deployment, CI/CD pipeline behaviours, ML pipeline behaviours and ML-specific component architectures [4], [5], [167] must be considered. Compounding the complexity of MLOps systems are the myriad tools and technologies available [4], [5], [26], [230].

Whilst the introduction of MLOps is welcome, the complexity of MLOps systems as described above and the variety of practices [23], [24], [56] make it challenging to determine which ADDs are applied in a given solution. The multitude of supporting technologies and programming languages makes it even more challenging to achieve this programmatically and generically, primarily through source code parsing. It is just as challenging to determine which decision options an architect selected, which specific patterns and practices were applied and the extent of support for core architectural quality attributes, such as automation, model management and simplified model deployment within the systems. Within MLOps systems, as exemplified in this study, implementing an automated assessment method would enhance understanding of a system's overall properties and their alignment with preferred practices and desired qualities.

The understanding outlined above is particularly pertinent to automation in MLOps systems, as the level of automation is directly influenced by practices related to integration and delivery, such as data processing and pipeline triggering, especially within an automated CD framework [22]. MLOps systems that support automation help streamline and accelerate the deployment and management of ML models in production environments. Teams can iterate quickly, maintain consistency and ensure reliability in ML workflows. Automation also reduces manual effort, minimises errors and enhances efficiency, allowing organisations to deploy and maintain ML models at scale whilst improving time-to-market and overall productivity.

In previous work described in this doctoral thesis [23], [24], [131], we identified several ADDs along with corresponding decision options and their respective decision drivers (which are synonymous with, or closely related to, quality aspects). In those studies, we achieved theoretical saturation, a stage in a Grounded Theory study at which the collection and analysis of further data no longer yield new theoretical insights. It is important to note that the work described in this chapter is not based solely on our prior work, and that our findings from those studies were corroborated in related work (see Section 8.2). Indeed, we have studied much scientific and practitioner literature from other authors [4], [25], [28], [29], [30], [56], [230], [231] since our initial studies were published, and these studies have corroborated our original findings based on grey literature. Note that a renewed search of practitioner and scientific literature before commencing this study did not yield any new insights concerning ML ADDs. Therefore, we concluded that no further extensive literature review was needed.

Several crucial quality attributes for ML systems that are essential for ensuring their effectiveness and robustness include performance efficiency (which strives for optimal resource use), maintainability (which focuses on ease of modification for improvements), scalability and portability (which address a system's ability to adapt to different capacities

and environments) and interoperability (which ensures seamless information exchange between systems). Process and work automation in ML is also an important quality attribute – automation enhances consistency, reduces human error and accelerates the deployment of ML models. Key automated processes include CI/CD pipelines, automated testing and model monitoring. These processes ensure that ML models are consistently validated, updated and maintained without manual intervention, thereby improving reliability and efficiency in production environments.

Automation thus plays a critical role in maintaining high quality and operational excellence in MLOps systems. Together, these attributes help in developing high-quality ML systems with robust architectural designs in MLOps. Note that we are not assuming any particular automation concept, paradigm or set of predefined practices. In this study, we modelled and manually assessed 22 system architectures, each with its own level of automation and set of practices (see Section 8.3.2).

The ADDs selected for this study are a subset of those identified in the aforementioned work, specifically those associated with automation-related factors, as the study focuses on the automation-related aspects of ML architectures and automation-relevant practices. The ADDs under consideration are *How to Automate Integration and Delivery in an ML Context?*, *How to Trigger an ML Pipeline or Orchestrator?* and *How to Automatically Process the Data Used for Model Building?*, and are described in detail in Section 8.4 together with their respective decision options.

The ADDs are derived from established practices in grey literature, such as informal practitioner guidelines, blog posts, public repositories and similar sources. Building upon this architectural knowledge, we aim to automatically evaluate the degree of architectural conformance to automation-relevant practices within modelled MLOps system architectures. Thus, this chapter aims to explore whether it is possible to determine the level of support for automation as a central quality aspect within MLOps systems.

This chapter is based on **P7**, presents **RC7** and its focus is to address the following research questions in the context of **RQ4** and **RP4**:

RQ4.1 *How can MLOps systems' support for the core quality aspect of automation be assessed in the context of ADDs and their respective decision options?*

RQ4.2 *What types of metrics can be applied to evaluate these levels of support, and how effective are they?*

To tackle these research questions, we introduced a set of metrics designed for diverse ADDs related to automation, encompassing all known decision options. We describe a model-driven, metrics-based methodology that relies solely on modelling elements obtained from MLOps systems, architectural descriptions, reference architectures or industrial architecture guides. These quantifiable system models encapsulate the architecture, properties and behavioural aspects of systems, thereby facilitating the evaluation of support for ADDs and enabling automated measurement of quality in MLOps system architectures via custom metrics.

Our approach involves establishing a ground truth through the manual evaluation of a set of system models, gauging their adherence to the full spectrum of considered

decision options and their combinations. Specifically, the pertinent ADDs and their associated decision options focus on *Integration and Delivery*, *Pipeline Triggers* and *Data Processing*. Ground truth creation entails objectively assessing the presence and level of support for each decision option within a system.

Implementing detectors in Python to analyse modelled systems and calculate our custom metrics, we assess the effectiveness of these metrics in predicting the ground truth using ordinal regression analysis. We demonstrate that simple yet effective metrics offer reliable insights, particularly in the context of automation, which is crucial for practitioners. Our prediction models demonstrate remarkable accuracy in predicting the ground truth assessment. Our approach can be automated within a CI/CD pipeline to help identify complex architectural features, such as ADDs and decision options, to streamline development cycles and ensure compliance with non-functional requirements.

The core contributions of this chapter are: (1) a novel approach for the semi-automated, model-driven and metrics-based statistical assessment of conformance of MLOps systems to automation-related ADDs; (2) detailed modelling of 22 MLOps system architectures and the continued development of a reusable MLOps metamodel; (3) the definition and application of novel, technology-agnostic, automation-specific metrics for assessing MLOps systems. In contrast to our prior work on ADDs, which identified ML workflow, ML deployment and RL training strategy-related ADDs, this chapter considers automation-related ADDs for ML and their assessment via metrics, to assess the quality of MLOps systems in terms of automation support.

A novel contribution of our study is the systematic approach we employed to source, model and assess various MLOps system architectures. Whereas our prior work mainly focused on the discovery of ML and RL-related ADDs, or modelled only certain aspects of very few systems, this study comprehensively models several MLOps system architectures in great detail and develops an extensive MLOps metamodel. This work makes a contribution to the field of MLOps architecture by introducing technology-agnostic metrics to evaluate support for automation-related ADDs. We empirically validated the metrics by performing ordinal regression analysis to develop reliable, reusable prediction models. In contrast, our prior work on ADDs was more qualitative than quantitative.

This chapter is organised as follows: Section 8.2 compares related work, whilst Section 8.3 outlines our research methods, ground truth definition and the system modelling approach. Section 8.4 details the ADDs and associated decision options considered in the study. The assessment process for the ground truth is explained in Section 8.5. Moving forward, Section 8.6 introduces our proposed metrics and Section 8.7 presents the results of the metric calculations and ordinal regression analysis. Section 8.8 discusses our research questions, a practical application of our approach and potential threats to validity. Future work is discussed in Section 8.9, and Section 8.10 provides concluding remarks.

8.2 Related Work

This section examines existing research on ML patterns, practices, quality aspects and metrics, and compares it with our study.

8.2.1 Patterns, Practices and Their Relation to Quality Aspects

ML and MLOps systems have grown in complexity and become more prevalent within the industry. Consequently, more scientific research is being conducted to catalogue and document quality aspects, as well as related patterns and practices. Patterns and practices are relevant to ADDs because they represent specific decision options, which are then associated with decision drivers (forces) that represent quality aspects of interest to practitioners, helping to guide them to the most suitable design decisions.

Sharma and Davaluri [123] detected and analysed design and architectural patterns in two ML applications. They identified various quality attributes, including reusability, flexibility and testability, which are critical for the sustainable development and maintenance of ML systems.

Washizaki, Uchida et al. [232] analysed ML systems' engineering, architecture, design patterns and anti-patterns. They suggest that improving crucial ML quality aspects such as complexity, performance, reliability and ML model quality can be achieved through best practices.

Washizaki, Khomh et al.'s [122] multivocal literature review included an ML practitioner survey. They identified various software engineering design patterns for ML applications. They propose that adopting these patterns could enhance system and software quality attributes, such as usability, reliability and maintainability, as well as ML quality attributes, such as ML model robustness, explainability and fairness.

Kreuzberger et al. [4] conducted mixed-methods research, including a literature review, tool assessment and expert interviews, providing a comprehensive overview of diverse ML aspects, such as components, architecture and workflows. They connect principles and practices, exploring the automation potential of manual ML processes and highlighting the importance of automation in MLOps.

Faubel and Schmid [56] conducted a pilot case study assessing MLOps applications in the industry. They observed MLOps practices, architectural patterns, and diverse types of automation across the surveyed companies. They noted commonalities with aspects relevant to this study, including pipeline triggers, CI/CD automation, pipeline automation, scripts and orchestrators.

A systematic literature review conducted by Lima et al. [231] focused on practices, patterns, roles, maturity models, challenges, and tools for automating operational activities in ML model deployment. Emphasising the significance of automation in deploying ML models, they recognised that existing maturity models indirectly gauge activities in ML model development. However, they highlighted the lack of direct assessments of MLOps, suggesting an opportunity for further research.

The cited related works collectively explore various facets of ML and MLOps systems, often highlighting practices and patterns similar to those considered in this study, as well as their relation to quality aspects. They acknowledge the importance of automation in MLOps systems, aligning with our understanding of automation as a core quality attribute. Automation streamlines processes, reduces human error, and enhances efficiency – all critical factors in MLOps systems. Moreover, automated pipelines and workflows enable scalability, allowing organisations to handle increasingly large and

complex datasets and models.

Our approach differs in focus and methodology from the related work described previously, as we leverage our knowledge of patterns and practices, and consider automation as a quality attribute to develop a model-driven, metrics-based approach for assessing support for automation in the context of ADDs in MLOps systems. By integrating automation assessment directly into the architectural design and development phases, quality assurance for automation can be applied from the outset, ensuring that MLOps systems are designed with automation in mind. This proactive approach reduces the need to retrofit automation and quality assurance into existing systems, reducing technical debt and accelerating time-to-market for ML solutions.

Furthermore, our metrics-based approach provides quantifiable measures of automation support, enabling stakeholders to make informed decisions about the trade-offs between automation and other quality attributes. By quantifying automation readiness at the architectural level, organisations can prioritise investments in automation infrastructure and tooling, strategically allocating resources to maximise automation benefits whilst mitigating associated risks. Ultimately, our study extends beyond acknowledging the importance of automation in MLOps systems to providing an approach for evaluating and enhancing automation support as a first-class quality attribute.

8.2.2 ML Metrics

There is a distinct lack of studies that quantitatively evaluate architectural aspects of ML and MLOps systems. However, authors have explored using runtime metrics to assess crucial aspects of ML and MLOps systems and ML models.

Cardoso Silva et al. [233] emphasise the significance of factors such as task completion time, CPU and GPU usage, memory usage, disk input/output and network traffic in MLOps. They propose an ML benchmarking approach to monitor solutions in production, aiding in decision-making, resource allocation and the operation of ML systems.

Y. Zhou et al. [31] created a functional DevOps-capable ML platform. They built and executed ML pipelines with diverse layers and hyperparameters, collecting and evaluating metrics for ML pipeline steps, the ML platform and ML models. The study identified potential performance bottlenecks, such as GPU utilisation, offering a valuable practical reference for constructing ML pipeline platforms.

J. Zhou et al. [234] conducted a survey providing an overview of methods proposed in the literature for assessing ML model explanations. They identify explainability as a target for quantitative metrics and discuss qualitative, human-centric metrics.

A literature review conducted by Carvalho et al. [235] identifies established methods and metrics for interpreting ML models. It highlights valuable work in interpretability assessments and recommends future research areas.

The acknowledged works are relevant to our study as they assess various quality aspects of ML systems and ML models. However, they focus on runtime performance metrics rather than system architecture metrics and do not specifically evaluate ML system support for automation-related ADDs. Whilst runtime metrics are crucial for understanding the operational efficiency and performance of ML systems during execution, they do not

provide insights into the underlying patterns and practices that influence automation capabilities. Our approach is model-driven and metrics-based, assessing automation support in MLOps architectures by incorporating architectural quality metrics into our analysis.

To our knowledge, no prior studies have considered architectural quality metrics for modelled MLOps systems, so we are initiating a new research direction in this field. By leveraging architectural modelling techniques and novel quality metrics, we can systematically evaluate the readiness of MLOps architectures for automation and identify opportunities for improvement.

8.3 Research Method

In this section, we outline our overall research process, detail our MLOps system search and selection methodology and give an overview of our ADD ground truth assessments, system modelling method and detectors. Figure 8.1 provides an overview of our research process and is described below. All artefacts are available in our replication package [236].

8.3.1 Contributions from Prior Work

Our study makes use of results and contributions from previous work [23], [24], where we performed a *Grey Literature Search* and *Grounded Theory Analysis* [65], [76], [96] through iterative consideration of practitioner literature. This process enabled us to *Formulate Core ADDs* for ML, the ones of interest for our study being automation-related (see Section 8.4).

As outlined in Section 8.1, our previous findings [23], [24] were corroborated by subsequent scientific literature related to the subject. Our methodological choices were influenced by the nature of the literature we analysed. To obtain industry-relevant insights, we employed a deductive approach in that prior work based on Straussian Grounded Theory, which is well-suited to an unbiased exploration of ML-related phenomena grounded in first-hand practitioner experience, especially since we found no prior ML-related Grounded Theory studies when searching for related work. For the white literature (peer-reviewed articles), we adopted an inductive approach to ensure our initial findings aligned with those of other researchers. The combination of these approaches provided us with a confident and nuanced understanding of ML-related practices, particularly in the context of MLOps.

We also devised an *Initial MLOps System Modelling* method [69], described earlier in the doctoral thesis, which we utilise in this study. Replication packages containing relevant artefacts from our prior studies are available in long-term repositories [103], [112], [185].

8.3.2 Core Activities and Contributions of this Work

Search Web Resources for Candidate Systems Our first step involved using popular topic portals (InfoQ and DZone), search engines (Google, DuckDuckGo and Bing) and GitHub source code repositories to identify relevant MLOps systems and their descriptions. Our search terms included "MLOps implementation", "MLOps system architecture" and "MLOps system example".

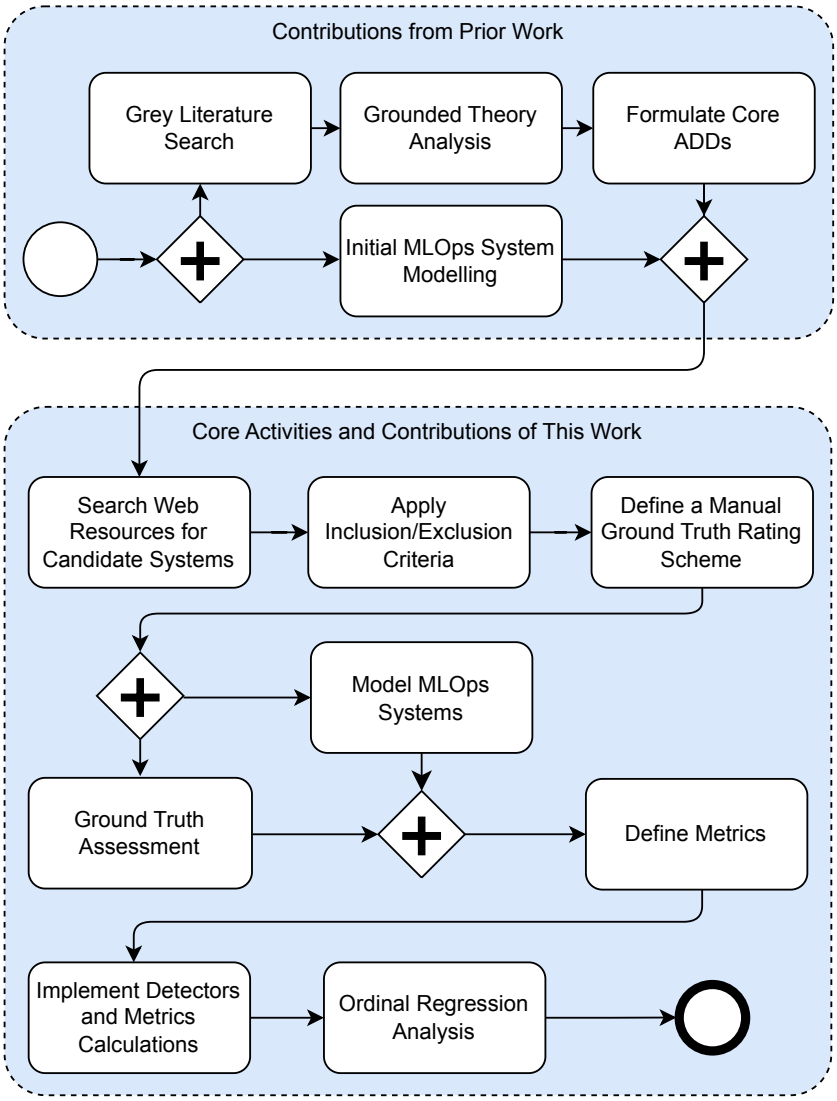

Figure 8.1: Overview of the research process we followed in this study.

Apply Inclusion/Exclusion Criteria The next step involved deciding which MLOps architectures resulting from our search to include in our study based on predefined inclusion-exclusion criteria. If a resource met one of the inclusion criteria, we then checked the exclusion criteria. Conversely, if the resource did not meet any of the inclusion criteria, we immediately excluded it. If an exclusion criterion was immediately apparent, then we checked neither the inclusion criteria nor any further exclusion criteria. Of the 32 candidates we identified, only nine met our inclusion criteria. For replicability and traceability, this stage of the research process is included in our replication package [236].

We defined our **inclusion criteria** as follows:

- MLOps systems: we considered systems that provide end-to-end solutions for training, deployment and management of ML models in production environments.
- Architectural descriptions: we included sources that provide detailed descriptions of the architecture and design of MLOps systems.
- Reference architectures: we included reference architectures that serve as blueprints or guidelines for building MLOps systems.
- Industrial architecture guides: we considered guides and best practices for architecting MLOps solutions in an industrial setting.

We established the following **exclusion criteria**:

- Student and pet projects: we omitted projects made by students for educational purposes or personal experiments. We categorised pet projects as GitHub repositories with fewer than 100 stars and 40 forks, suggesting low acceptance and maturity.
- Experimental or proof of concept examples: we excluded sources that discuss or implement technologies unsuitable for real-world industrial use. These include systems that lack production-ready functionality, scalability or dependability.
- Tools and libraries: we omitted sources that only discuss or implement specific tools or libraries used in the ML workflow, such as data preprocessing libraries or model training frameworks, as these sources do not provide a comprehensive overview of MLOps architectures as a whole.
- Specific parts of the ML workflow: we excluded sources that do not address the entire ML workflow, such as those that focus only on data preparation or model deployment, as they do not encompass the complete MLOps architecture.
- Advertisements: we omitted sources that primarily promote commercial items or services, as they may not provide objective architectural information.
- Lack of practical architectural examples: we excluded sources that did not provide realistic examples of MLOps architectures applicable to an industrial setting.
- Marketing: we omitted sources that discuss consultancy firms' techniques, as they may not have reflected general real-world MLOps designs.

Define a Manual Ground Truth Rating Scheme In subsequent steps, we established a ground truth rating scheme for MLOps systems to assess their support for automation-related ADDs.

Ground Truth Assessment/Model MLOps Systems We performed a manual ground truth assessment as described in Section 8.5 and concurrently systematically modelled MLOps system architectures and their variations, ensuring comprehensive coverage of the design space. We utilised an MLOps system metamodel and followed a systematic modelling approach [69] to construct formal architectural models. These models, implemented in Python using Codeable Models [78] and visualised as UML diagrams using PlantUML [79], comprise nodes and connectors representing components and relationships.

The modelling process resulted in 22 models based on real-world MLOps systems from vendors that provide computing services to ML application developers, including Alibaba Cloud, Amazon Web Services, Google, Microsoft and Red Hat. We assigned each modelled system an ID: base systems, whose IDs always end with a 1, include AP1, MA1 and AC1 and variants of the base systems, whose IDs always end with a number other than 1, such as AP2, MA2 and AC2. Table 8.1 lists the modelled systems, the size of each model in terms of how many nodes, connectors and pipelines[1], a description of each system and the URL of the source of each base system.

Table 8.1: Included MLOps System Architectures: IDs, Model Sizes, Descriptions and Source URLs

System	Model Size	System Description
AP1	18 nodes, 25 connectors, 1 pipeline	Guide to building an automated MLOps pipeline using Amazon AWS CodePipeline, CodeBuild, SageMaker, Lambda, S3, the AWS container registry and Docker [33] (source: `https://tinyurl.com/mlops-system-ap`).
AP2	21 nodes, 25 connectors, 2 pipelines	Variant of AP1 with partial support for automated *Build and Deployment Scripts*, advanced triggers but no basic triggers, and support for data processing via a *Data Pipeline*.
AP3	19 nodes, 22 connectors, 2 pipelines	Variant of AP1 with support for *Build and Deployment Scripts*, a single basic trigger and no data processing automation.
MA1	55 nodes, 58 connectors, 3 pipelines	Reference architecture for a Microsoft Azure project that demonstrates how to automate an end-to-end ML/AI workflow (source: `https://tinyurl.com/mlops-system-ma`).
MA2	53 nodes, 56 connectors, 3 pipelines	Variant of MA1 with partial support for integration and delivery automation via *CI/CD Pipelines*, no basic trigger support but support for an advanced trigger, and partial data processing automation via an *ETL Pipeline*.

Continued on next page...

[1] See our replication package [236] for the source code and model diagrams.

System	Model Size	System Description
MA3	24 nodes, 24 connectors, 1 pipeline	Variant of MA1 with no integration and delivery automation, support for a single advanced trigger but no basic triggers, and data processing automation support via a *Data Pipeline*.
AC1	33 nodes, 67 connectors, 1 pipeline	Reference architecture and workshop/tutorial to implement an Amazon AWS MLOps pipeline that automates the typical procedures used by ML practitioners (source: `https://tinyurl.com/mlops-system-ac`).
AC2	31 nodes, 63 connectors, 1 pipeline	Variant of AC1 with support for *Build and Deployment Scripts*, a single advanced trigger but no basic trigger, and partial support for data processing automation via a *Data Processing Component*.
AC3	36 nodes, 71 connectors, 1 pipeline	Variant of AC1 with support for an *ML Orchestrator*, full trigger support, and partial support for data processing automation via an *ETL Pipeline*.
GV1	78 nodes, 137 connectors, 3 pipelines	Google Cloud Platform implementation of MLOps with Vertex AI, Smart Analytics, Keras, TFX and Model Builder SDK (source: `https://tinyurl.com/mlops-system-gv`).
GV2	54 nodes, 96 connectors, 1 pipeline	Variant of GV1 with no integration and delivery automation, no automation via triggers, but support for data processing automation via a *Data Pipeline*.
GV3	70 nodes, 126 connectors, 3 pipelines	Variant of GV1 with partial *ML Orchestrator* support, support for a single basic trigger, and partial support for data processing automation via a *Data Processing Component*.
AS1	35 nodes, 55 connectors, 1 pipeline	Alibaba Cloud repository and article focusing on the ML pipeline with MLOps using Serverless Workflow, Function Compute, and Container Service for Kubernetes provided by GitHub (source: `https://tinyurl.com/mlops-system-as`).
AS2	36 nodes, 57 connectors, 1 pipeline	Variant of AS1 with partial support for an *ML Orchestrator*, support for almost all triggers, and support for data processing automation via a *Data Processing Component*.
AS3	34 nodes, 56 connectors, 1 pipeline	Variant of AS1 with partial support for all three integration and delivery automation options, full trigger support, and support for data processing automation via a *Data Processing Component*.

Continued on next page...

System	Model Size	System Description
RO1	34 nodes, 61 connectors, 1 pipeline	MLOps architecture description using Red Hat OpenShift, applying DevOps and GitOps principles (source: `https://tinyurl.com/mlops-system-ro`).
RO2	29 nodes, 53 connectors, 1 pipeline	Variant of RO1 with partial *CI/CD Pipeline* support, support for a single advanced trigger, but no basic triggers, and no data processing automation support.
RO3	19 nodes, 32 connectors, 0 pipelines	Variant of RO1 with no automation support.
GM1	18 nodes, 16 connectors, 1 pipeline	Google Cloud Architecture Center description of an MLOps architecture at the basic level of maturity, where building and deploying ML models is entirely manual (source: `https://tinyurl.com/mlops-system-gm`).
GA1	55 nodes, 69 connectors, 2 pipelines	Describes an architecture with continuous model training via an automated ML pipeline, automated retraining on new data, automated data and model validation, pipeline triggers and metadata management to achieve continuous delivery of a model prediction service (source: `https://tinyurl.com/mlops-system-ga`).
GC1	64 nodes, 77 connectors, 3 pipelines	Architecture for a robust, automated CI/CD system for building, testing, and deploying new pipeline components to a target environment (source: `https://tinyurl.com/mlops-system-gc`).
GC2	64 nodes, 77 connectors, 3 pipelines	Variant of GC1 with similar support for integration and delivery automation, and automation via triggers, but slightly worse support for data processing automation via *Data Pipelines*.

MLOps system model views commonly include component and pipeline views. Component views depict nodes representing various elements such as execution environments, cloud components, platforms and connectors indicating their relationships, specifying details like artefact (e.g. ML model) provision, pipeline triggers, trigger data, i.e. the data associated with a trigger event, such as information on the trigger event itself or data relevant for the component or pipeline that was triggered, deployment targets (e.g. the components or environments where models will be deployed and executed to make decisions or predictions) and pipeline interactions and relations. Figure 8.2 depicts a component view of the modelled MLOps system AP1 (see Table 8.1).

Pipeline views display internal pipeline details, including an initial node, execution nodes and a final node. Fork nodes allow parallel execution that converges at a join node. They convey various aspects like triggers (described above), data processing (the stage in which raw data is transformed into a format that is suited for ML, like data sanitisation, feature engineering and normalisation), model training (the process by which a model learns from data), testing and evaluation (using a test dataset to assess how well a model

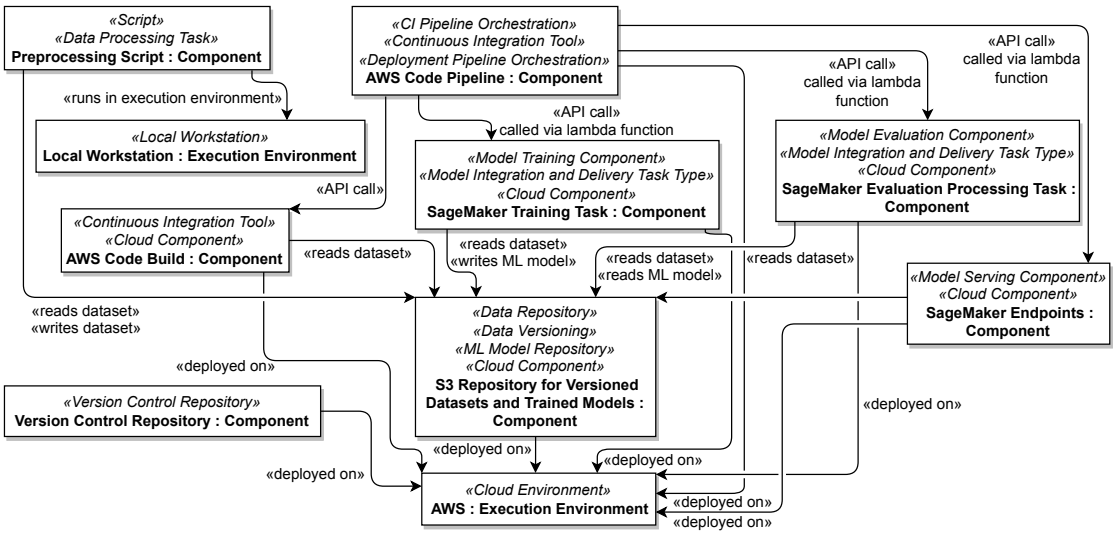

Figure 8.2: A Component UML model diagram for the MLOps system AP1.

generalises to unseen data), validation (tuning the hyperparameters of a model with a validation dataset and preventing overfitting) and deployment (described above), with metadata describing automation (e.g. whether a pipeline step runs automatically), and deployment or runtime environments (such as the specific component associated with the environment). Figure 8.3 illustrates a pipeline UML view of system AP1.

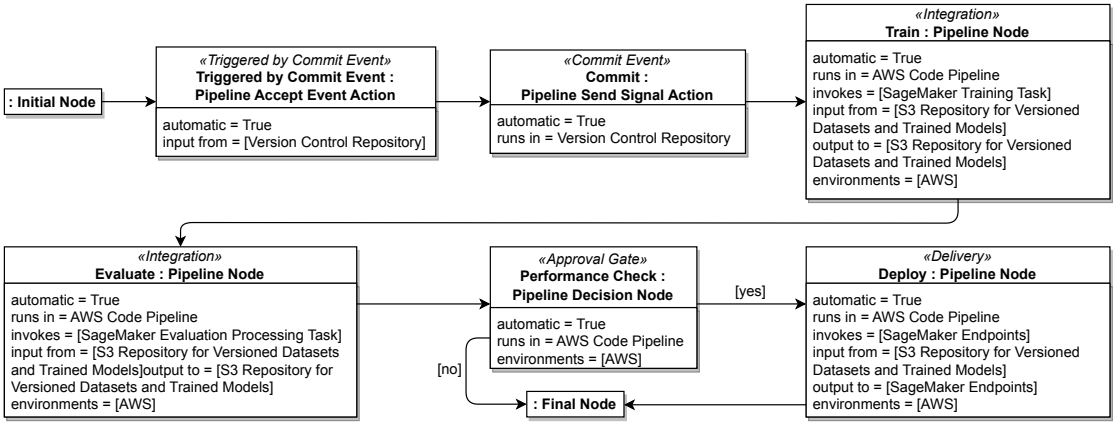

Figure 8.3: A deployment pipeline UML model diagram for the MLOps system AP1.

Define Metrics We defined objective, technology-agnostic metrics to measure conformance to previously identified automation-relevant decision options for ADDs. Our custom metrics are described in detail in Section 8.6.

Implement Detectors and Metrics Calculations Modelling MLOps system architectures enabled us to implement detectors that analyse model components, pipelines and their relations, automatically calculating custom metrics and supporting ordinal regression analysis. These detectors traverse the models, enabling the detection of metrics-relevant architectural properties, and offering a reusable approach for new systems and metrics. Whilst alternative approaches, such as checklists, could be applied rather than using our detectors, this alternative is less effective, less suited to ordinal regression analysis, and would require more manual effort during a system analysis than our automated metrics do.

Ordinal Regression Analysis Finally, we applied ordinal regression analysis, a common statistical approach, to assess how well the metrics predicted the previously established ground truth for the selected ADDs and modelled architectures. Our analysis and evaluation are described further in Section 8.7.

8.4 Core ADDs

This section introduces three automation-related ADDs and their associated decision options. We selected them from our prior work [23], [24] for their influence on MLOps automation. These ADDs encompass various decision drivers (forces), relations and practices found in grey literature. The following discussion on the relevance of these decision options on automation supports our rationale for the manual ground truth assessment detailed in Section 8.5.

8.4.1 How to Automate Integration and Delivery in an ML Context? ("Integration and Delivery")

When deciding *How to Automate Integration and Delivery in an ML Context*, choosing **No Integration or Delivery Automation** is the most straightforward option. However, numerous experts strongly discourage this approach, emphasising its adverse effects on various factors, notably *process and work automation*.

An alternative to continuously automating the deployment process is to utilise a **CI/CD Pipeline**. Our sources strongly endorse this option, especially for *process and work automation* and for facilitating *automated provisioning*, which pertains to the actual automation of the deployment step. In Figure 8.2, for instance, relevant components are *AWS Code Build* and *AWS Code Pipeline* and in Figure 8.3, the pipeline nodes from *Train* onwards are relevant.

Another approach involves executing automated deployments through **Build and Deployment Scripts**. Whilst this choice contributes to *process and work automation*, it may require manual intervention rather than achieving complete automation.

The **ML Orchestrator** represents another viable decision option. Employing an orchestrator enhances *process and work automation*, reducing the reliance on manual execution and minimising the potential for human error, thereby allowing engineers to focus on other tasks.

8.4.2 How to Trigger an ML Pipeline or Orchestrator? ("Pipeline Triggers")

ML pipelines and orchestrators can be initialised via triggers. Triggers exhibit diverse characteristics, and the decision options' forces for this ADD are inherited from higher-level decision options, for instance, whether a **CI/CD Pipeline**, **Build and Deployment Scripts** or **ML Orchestrator** was used. The trigger options themselves imply various levels of support for automation.

The trivial case is not to use an automated trigger, i.e. to use an **On-Demand Trigger** (manual execution by a human operator), which suggests no automation. This lack of automation allows the trigger to be omitted from the study. An **On Commit Trigger** activates components after a commit event in a source control management or data versioning repository system, thereby providing some degree of automation. For system AP1, for instance, this is visible in the *Triggered by Commit Event* pipeline accept event action in Figure 8.3. An **On Schedule Trigger** initiates a component at specific times or with defined regular intervals. An **On Availability of New Training Data Trigger** can lead to the triggering of a **CI/CD Pipeline**, **Build and Deployment Scripts** or **ML Orchestrator** when new training data becomes available. An **On Model Performance Degradation Trigger** can be used when ML model performance deteriorates, and an **On Changes to the Data Distribution Trigger** can be applied when statistical changes in any dataset used for training ML models surpass a predefined threshold.

8.4.3 How to Automatically Process the Data Used for Model Building? ("Data Processing")

A fundamental automation-related ADD pertains to data processing automation for building ML models. The trivial option is **No Data Processing Automation**. Three alternative decision options to the trivial case offer vital advantages to varying degrees, such as facilitating *automated data collection* and enabling a high level of *process and work automation*, and are described below.

Some practitioners advocate an **ETL Pipeline**. It entails extracting data from diverse sources, transforming it for analysis or storage and loading it into a database, data warehouse or data lake. The primary objective is to prepare the data for downstream analytics or applications, often by integrating and consolidating data from various sources to ensure quality and consistency. **ETL Pipelines** are typically batch-oriented and scheduled periodically for data upkeep. However, this approach is documented as being employed only occasionally in practice.

A prevalent choice amongst the automation architectures commonly suggested in the previously cited work is the use of a **Data Pipeline**. This approach involves conducting data processing within a dedicated, adaptable pipeline. Unlike **ETL Pipelines**, data pipelines encompass a wider range of operations beyond just extraction, transformation and loading. They can also include tasks such as data ingestion, data preprocessing and feature engineering. This decision option stands out as the most versatile amongst the automated choices.

Whilst pipeline architectures are frequently proposed, there are also suggestions for

Data Processing Components that do not adhere to a pipeline structure. Unlike an **ETL Pipeline**, a data processing component may not necessarily involve the extraction and loading of data and, in contrast to data pipelines, which encompass the entire flow of data, **Data Processing Components** instead focus on performing specific data processing tasks such as data transformation, feature engineering, data augmentation, scaling, and other preprocessing steps required for ML model training and analysis.

8.5 Ground Truth Rating Scheme and Assessment

This section provides insight into establishing ground truth data for each ADD outlined in Section 8.4.

We established the ground truth through a thorough, systematic and manual assessment of the selected systems, based on findings from grey literature in prior studies and multiple rounds of independent review and validation by all three members of the author team. We discussed any instances of disagreement in the ground truth assessment and refined it until we reached consensus. Thanks to our use of a high-level abstraction of MLOps systems according to an existing, extensible MLOps metamodel during the modelling process (see Section 8.3.2), it was straightforward for us to identify the elements relevant to the ground truth assessment collectively. We corrected any inconsistencies in both the metamodel and our ground truth assessment. We categorised the relevant decision options as either *supported*, *partially supported* or *not supported* (denoted by **S**, **P** or **N**, respectively in Table 8.2) or, for decision options that are simply present or absent, with Boolean values.

By combining the outcome of all decision options for each ADD and system, and informed by the descriptions from the literature of the force impacts of the various decision options in Section 8.4, we then derived an ordinal assessment of how well the decision as a whole was supported in each modelled system, according to the following ordinal scale:

- ++: Automation is very well supported.
- +: Automation is well supported, but aspects of the system could be improved to better support automation.
- **o**: Aspects of the system need to be improved concerning automation, but substantial support for automation is present.
- **-**: Aspects of the system need to be improved concerning automation, but some support for automation is present.
- **--**: No support for automation is present.

Whilst our ordinal scale of **--** to ++ adheres to the given order, it does not assume equal intervals, unlike a Likert scale. The specific assessment criteria for each ADD are described in Sections 8.5.1, 8.5.2 and 8.5.3, and we recorded the resulting ordinal assessments in the ***Assessments*** rows of Table 8.2.

Consider system AP1 in Table 8.2, for instance. For the *Integration and Delivery* ADD, AP1 does not support the decision option **Build and Deployment Scripts**

Table 8.2: Ground Truth Data for ADDs, Decision Options and the Included MLOps System Architectures

ADDs/Decision Options	**AP1**	**AP2**	**AP3**	**MA1**	**MA2**	**MA3**	**AC1**	**AC2**	**AC3**	**GV1**	**GV2**	**GV3**	**AS1**	**AS2**	**AS3**	**RO1**	**RO2**	**RO3**	**GM1**	**GA1**	**GC1**	**GC2**
Integration and Delivery																						
Build and Deployment Scripts	N	P	S	N	N	N	N	S	N	N	N	N	N	N	P	N	N	N	P	N	N	N
CI/CD Pipeline	S	N	N	P	P	N	S	N	N	S	N	N	N	N	P	P	P	N	N	S	S	S
ML Orchestrator	N	N	N	P	N	N	N	N	S	N	N	P	S	P	P	N	N	N	N	P	P	P
Assessments	++	-	o	+	-	--	++	o	+	++	--	+	+	-	o	+	-	--	--	++	++	++
Pipeline Triggers																						
On Commit	T	F	F	T	F	F	T	F	T	T	F	F	T	T	T	T	F	F	F	F	T	T
On Schedule	F	F	F	T	F	F	F	F	T	T	F	T	F	T	T	F	F	F	F	T	T	T
Availability of New Training Data	F	T	F	T	F	F	T	T	T	F	F	F	F	T	T	F	F	F	F	T	T	T
Model Performance Degradation	F	T	F	T	F	F	F	F	T	F	F	F	F	F	T	T	T	F	F	T	T	T
Changes to the Data Distribution	F	T	F	F	T	T	F	F	T	F	F	F	F	T	T	F	F	F	F	T	T	T
Assessments	o	-	--	+	-	-	+	-	++	o	--	o	o	+	++	+	-	--	--	++	++	++
Data Processing																						
Data Pipeline	N	S	N	N	N	S	N	N	N	N	S	N	N	N	N	N	N	N	N	P	P	P
ETL Pipeline	N	N	N	N	P	N	S	N	P	N	N	N	N	N	N	N	N	N	N	N	N	N
Data Processing Component	N	N	N	N	N	N	N	P	N	P	N	P	S	S	S	N	N	N	N	N	N	N
Assessments	--	++	--	--	-	++	+	o	-	o	++	o	+	+	+	--	--	--	--	+	+	-

(category **N**), it supports the decision option **CI/CD Pipeline** (category **S**), and it does not support the decision option **Machine Learning Orchestrator** (category **N**). With reference to the scoring scheme defined in Section 8.5.1, this system receives a score of ++ for this ADD since the decision option **CI/CD Pipeline** is supported and ***all*** *integration and delivery tasks are automated and take place within a* ***CI/CD Pipeline***. This score also corresponds to the generic scoring scheme defined above, where ++ denotes that *automation is very well supported.*

After establishing and assessing the ground truth data, we evaluated how well the novel metrics defined in Section 8.6 predicted it using ordinal regression. We describe the ordinal regression analysis in Section 8.7.

8.5.1 Integration and Delivery

In Section 8.4.1, we discussed the level of support for automation for each *Integration and Delivery* option. On this basis, we developed the following scoring framework for our ground truth assessment of this decision:

- ++: **All** integration and delivery tasks are automated and take place within a **CI/CD Pipeline**.
- +: The **majority** of (but not all) integration and delivery tasks are automated and take place within a **CI/CD Pipeline** OR **all** integration and delivery tasks are automated and take place via an **ML Orchestrator**.
- **o**: The **majority** of (but not all) integration and delivery tasks are automated and take place via an **ML Orchestrator** OR the **majority or all** integration and delivery tasks are automated and take place within **Build and Deployment Scripts**.
- **-**: **Some** (but not the majority or all) integration and delivery tasks are automated and take place within a **CI/CD Pipeline**, via a **Build and Deployment Scripts** or within **Build and Deployment Scripts**.
- **--**: **No** integration and delivery task automation is supported.

8.5.2 Pipeline Triggers

Drawing upon the reasoning provided in Section 8.4.2 in support of the *Pipeline Triggers* decision, we devised a scoring scheme for our ground truth assessment:

- ++: **At least one** basic trigger (**On Commit Trigger** or **On Schedule Trigger**) is supported and **all** applicable advanced automatic triggers are supported (**On Availability of New Training Data Trigger**, **On Model Performance Degradation Trigger** and **On Changes to the Data Distribution Trigger**).

- +: **At least one** basic trigger (**On Commit Trigger** or **On Schedule Trigger**) is supported and **at least one** advanced automatic trigger is supported (**On Availability of New Training Data Trigger**, **On Model Performance Degradation Trigger** or **On Changes to the Data Distribution Trigger**).
- **o**: **At least one** basic trigger (**On Commit Trigger** or **On Schedule Trigger**) is supported, but **no** advanced triggers are supported.
- **-**: **At least one** advanced trigger is supported (**On Availability of New Training Data Trigger**, **On Model Performance Degradation Trigger** or **On Changes to the Data Distribution Trigger**), but **no** basic triggers are supported (**On Commit Trigger** or **On Schedule Trigger**).
- **--**: **No** automatic triggers are supported.

8.5.3 Data Processing

Lastly, based on the forces we described in Section 8.4.3 for the *Data Processing* decision, we established the following scoring scheme for our ground truth assessment:

- ++: **All** model-building-relevant data processing tasks are automated and occur within a **Data Pipeline**.
- +: **The majority** of model-building-relevant data processing tasks are automated and occur within a **Data Pipeline** OR **all** model-building-relevant data processing tasks are automated and occur within an **ETL Pipeline** or **Data Processing Component**.
- **o**: **The majority** of model-building-relevant data processing tasks are automated and occur within an **ETL Pipeline** or **Data Processing Component**.
- **-**: **Some** model-building-relevant data processing tasks are automated and occur within a **Data Pipeline**, an **ETL Pipeline** or a **Data Processing Component**.
- **--**: **No** data processing automation is supported.

8.6 Metrics

This section presents the metrics that we propose for each decision option discussed in Section 8.4. Following our iterative manual assessment and modelling of the MLOps system architectures from Table 8.1 to establish the ground truth ratings described in Section 8.5 and summarised in Table 8.2, we defined at least one metric per ADD decision option to measure their support for automation. In Section 8.7, we describe how we performed ordinal regression analysis to assess how well the custom metrics defined further below in this section predicted the ground truth data.

The metrics have been deliberately designed to be straightforward, allowing them to represent each decision effectively. They may be either a Boolean value indicating

the absence or presence of a given decision option within the respective system or a continuous value in $[0, 1]$, with 0 signifying the worst-case scenario where there is no support for the decision option and 1 symbolising an ideal scenario in which the decision option receives full support.

8.6.1 Metrics for the Integration and Delivery Decision

The ratio of components that support the **Build and Deployment Scripts** option is calculated by the **BDS** metric of Equation 8.1.

$$BDS = \frac{\begin{array}{c}|integration\ and\ delivery\ tasks \\ in\ build\ and\ deployment\ scripts|\end{array}}{|integration\ and\ delivery\ tasks\ in\ the\ system|} \tag{8.1}$$

The proportion of components enabling the **CI/CD Pipeline** option is revealed by the **CID** metric of Equation 8.2.

$$CID = \frac{\begin{array}{c}|integration\ and\ delivery\ tasks \\ automated\ in\ a\ CI/CD\ pipeline|\end{array}}{|integration\ and\ delivery\ tasks\ in\ the\ system|} \tag{8.2}$$

The fraction of components supporting the **ML Orchestrator** option is measured by the **MLO** metric of Equation 8.3.

$$MLO = \frac{\begin{array}{c}|integration\ and\ delivery\ tasks \\ automated\ in\ an\ ML\ orchestrator|\end{array}}{|integration\ and\ delivery\ tasks\ in\ the\ system|} \tag{8.3}$$

8.6.2 Metrics for the Pipeline Triggers Decision

We devised the following *Pipeline Triggers* metrics:

- The existence of the **On Commit Trigger** option is signified by the **COT** metric, which yields a Boolean value.
- Presence of the **On Schedule Trigger** option is conveyed by the Boolean **SCT** metric.
- Whether the **On Availability of New Training Data Trigger** option is represented in the system is signified by the **TDT** metric, also a Boolean.
- An indicator for the presence of the **On Model Performance Degradation Trigger** option is the Boolean **MPT** metric.
- The availability of the **On Changes to the Data Distribution Trigger** option is informed by the Boolean **DDT** metric.

Equation 8.4 thus gives the value for any of the above metrics:

$$COT, SCT, TDT, MPT, DDT = \begin{cases} True: & \text{if } \textit{the model contains} \\ & \textit{at least one relation of} \\ & \textit{the relevant trigger type;} \\ False: & \textit{otherwise.} \end{cases} \tag{8.4}$$

8.6.3 Metrics for the Data Processing Decision

The proportion of components in favour of the **Data Pipeline** option is indicated by the **DAP** metric of Equation 8.5.

$$DAP = \frac{|\textit{data processing tasks automated in a data pipeline}|}{|\textit{data processing tasks in the system}|} \tag{8.5}$$

The share of components supporting the **ETL Pipeline** option is revealed by the **ETP** metric of Equation 8.6.

$$ETP = \frac{|\textit{data processing tasks automated in an ETL Pipeline}|}{|\textit{data processing tasks in the system}|} \tag{8.6}$$

The ratio of components that are aligned with the **Data Processing Component** option is shown by the **DPC** metric of Equation 8.7.

$$DPC = \frac{|\textit{data processing tasks automated in a data processing component}|}{|\textit{data processing tasks in the system}|} \tag{8.7}$$

When applying the methodology described in this chapter, an architect familiar with the system architecture would be involved in classifying system features, interpreting metrics and setting thresholds. A lower value is considered worse than a higher value, depending on how desirable an architect deems supporting a given decision option. In this context, please note that the system's modularity does not directly influence the value of the metrics; instead, it is the nature of the components and their contained tasks, as well as the trigger types, that matter.

8.7 Ordinal Regression Analysis

Ordinal regression models the relationship between an ordinal dependent variable and one or more independent predictors. It helps understand the odds of an observation belonging to a specific category, assess the strength and direction of relationships, evaluate predictor significance and build accurate regression models for predictions with new data. The metrics expounded upon in Section 8.6 and summarised in Table 8.4 serve as the

independent predictor variables and are either Boolean or continuous values measured within the interval $[0, 1]$.

The results of the ordinal regression analysis[2] concerning each ADD are showcased in Table 8.3. The outcome variables, which depend on the ground truth assessments (ordinal variables) for each decision, as elucidated in Section 8.5 and shown in Table 8.2, are integral to the analysis.

Table 8.3: Ordinal Regression Analysis Results for the Metrics

Statistical Measures	**Values**
Integration and Delivery	
Intercept: $\geq$ [-]	-0.4118
Intercept: $\geq$ [o]	-2.2673
Intercept: $\geq$ [+]	-3.5752
Intercept: $\geq$ [++]	-5.7906
Metric Coefficient (BDS)	2.9438
Metric Coefficient (CID)	5.8622
Metric Coefficient (MLO)	4.4388
Regression Model p-value	8.2844e-09
Regression Model Brier Score	0.0637
Regression Model C-index (original)	0.9895
Regression Model C-index (bias-corrected)	0.9192
Pipeline Triggers	
Intercept: $\geq$ [-]	-0.1809
Intercept: $\geq$ [o]	-2.2715
Intercept: $\geq$ [+]	-4.0512
Intercept: $\geq$ [++]	-6.4206
Metric Coefficient (COT)	2.4227
Metric Coefficient (SCT)	1.7147
Metric Coefficient (TDT)	1.1081
Metric Coefficient (MPT)	1.4322
Metric Coefficient (DDT)	1.0823

Continued on next page...

[2] When conducting the ordinal regression analysis, we utilised the *lrm* function from the *rms* package in R [237].

Statistical Measures	**Values**
Regression Model p-value	1.6090e-07
Regression Model Brier Score	0.0842
Regression Model C-index (original)	0.9585
Regression Model C-index (bias-corrected)	0.9363
Data Processing	
Intercept: $\geq$ [-]	-0.8649
Intercept: $\geq$ [o]	-2.1320
Intercept: $\geq$ [+]	-3.3447
Intercept: $\geq$ [++]	-6.4474
Metric Coefficient (DAP)	7.4840
Metric Coefficient (ETP)	4.2296
Metric Coefficient (DPC)	4.1724
Regression Model p-value	5.5376e-09
Regression Model Brier Score	0.05585
Regression Model C-index (original)	0.9946
Regression Model C-index (bias-corrected)	0.9595

The ordinal response variable is divided into increasing intercept categories, where each of $\geq$ [-], $\geq$ [o], $\geq$ [+], $\geq$ [++] as introduced in Section 8.5 represents the transition point of the corresponding level (i.e. for the dependent variable). The intercept coefficients in our regression models quantify the impact of each independent predictor on the outcome, indicating how each variable influences the odds of transitioning between response categories. Positive coefficients suggest an increased likelihood of moving to a higher category with each unit increase in the independent variable, whilst negative coefficients imply the opposite, indicating a reduced likelihood. For example, a coefficient of +5 corresponds to a five-fold increase in the dependent variable for every unit increase in the independent variable, and a coefficient of -30 signifies a 30-fold decrease.

The regression model C-index, also known as the concordance index or the area under the Receiver Operating Characteristic (ROC) curve, measures the discriminatory power [238] of a regression model. The original C-index assesses how well a regression model distinguishes between the categories or levels of an ordinal response variable. The bias-corrected C-index accounts for potential bias in a regression model. C-index values are continuous in the interval $[0, 1]$, with higher values indicating greater discriminatory power.

Harrell [237] recommends the utilisation of bootstrapping, a resampling technique, to

obtain nearly unbiased estimates of a regression model's future performance[3], i.e. its predictive power. A C-index of 1 represents perfect discrimination, whilst 0.5 indicates random chance. Regression models are considered good when the C-index exceeds 0.7 and strong when it surpasses 0.8. As illustrated in Table 8.4, all C-index values exceed 0.8, affirming our regression models' effectiveness in predicting individual outcomes.

The Brier score in ordinal regression assesses the quality of probabilistic predictions by measuring the mean squared difference between predicted probabilities and actual outcomes [239]. Unlike simple accuracy metrics, it offers a nuanced evaluation of calibration and accuracy. Lower scores signify better predictive accuracy, with 0 representing a perfect match. All three regression models demonstrate low Brier scores, indicating their ability to make accurate predictions.

In ordinal regression, p-values for regression model coefficients indicate the significance of predictors, usually with p-values less than or equal to 0.05 for statistical significance [240]. In Table 8.3, all p-values are less than 0.05, confirming the models' statistical significance and highlighting the substantial influence of included metrics on dependent variables.

Ordinal regression analysis is used for outcome variables with ordered categories that do not have equal distances between them, like the ones we defined. Using an overall metric based on average weight can violate the fundamental assumption of ordinal regression, as it would assume equal intervals between categories, which may not hold. Moreover, ordinal regression allows for modelling the probability of being in a particular category or lower based on a set of predictors. For this reason, we do not use averages of the various decision option metrics to derive an overall score for a modelled system, as this could oversimplify the complex relationships between predictors and outcome variables. Determining appropriate weights for the components of composite metrics is challenging, and relying on an average can fail to reflect each component's significance accurately. This issue is similar to those encountered with other metrics, such as the Maintainability Index [241]. Analysing each ADD separately and applying ordinal regression provides a more nuanced understanding of the data than averaging across them, as shown in Table 8.3. This approach yields reusable ordinal regression models, making it highly suitable for this study.

Our multi-metric approach ensures a robust assessment of regression model performance. All three models emerge as strong, statistically significant predictors of their respective ordinal response variables, collectively providing a comprehensive understanding of automation-related ADD support. Each model contributes valuable insights, collectively offering a well-rounded perspective.

8.8 Discussion

In this section, we address the research questions, discuss a practical application of our approach and examine potential threats to validity.

[3] We employed the *validate* function from the *rms* package to achieve bootstrapping and the subsequent calculation of the bias-corrected C-index.

Table 8.4: Metrics Calculation Results for the Included MLOps Architectures

	AP1	AP2	AP3	MA1	MA2	MA3	AC1	AC2	AC3	GV1	GV2	GV3	AS1	AS2	AS3	RO1	RO2	RO3	GM1	GA1	GC1	GC2
Integration and Delivery Metrics																						
BDS	0	0.333	1	0	0	0	0	1	0	0	0	0	0	0	0.667	0	0	0	0	0	0	0
CID	1	0	0	0.615	0.462	0	1	0	0	1	0	0	0	0	0	0.875	0.375	0	0	1	1	1
MLO	0	0	0	0.385	0	0	0	0	1	0	0	0.857	1	0.333	0	0	0	0	0	0.429	0.375	0.375
Pipeline Triggers																						
COT	T	F	F	T	F	F	T	F	T	T	F	F	T	T	T	T	F	F	F	F	T	T
SCT	F	F	F	T	F	F	F	F	T	T	F	T	F	T	T	F	F	F	F	T	T	T
TDT	F	T	F	T	F	F	T	T	T	F	F	F	F	T	T	F	F	F	F	T	T	T
MPT	F	T	F	T	F	F	F	F	T	F	F	F	F	F	T	T	T	F	F	T	T	T
DDT	F	T	F	F	T	T	F	F	T	F	F	F	F	T	T	F	F	F	F	T	T	T
Data Processing Metrics																						
DAP	0	1	0	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0.5	0.5	0.333
ETP	0	0	0	0	0.333	0	1	0	0.333	0	0	0	0	0	0	0	0	0	0	0	0	0
DPC	0	0	0	0	0	0	0	0.5	0	0.6	0	0.8	1	1	1	0	0	0	0	0	0	0

8.8.1 Discussion of the Research Questions

Our study addressed two key research questions regarding the assessment of automation support in MLOps systems, which we discuss below.

RQ$_{4.1}$: *How can MLOps systems' support for the core quality aspect of automation be assessed in the context of ADDs and their respective decision options?* To address **RQ$_{4.1}$**, we conducted a rigorous evaluation of MLOps systems' support for automation-related ADDs and the decision options they offer. Through an iterative modelling and evaluation process, we established a comprehensive ground truth, ensuring thorough coverage of the design space. Our technology-agnostic metrics assigned to each decision option, along with the subsequent ordinal regression analysis, demonstrated the effectiveness of our approach in objectively assessing the level of automation support in MLOps systems. This methodology ensures a thorough understanding of the nuances of automation-related ADDs and decision options within the context of MLOps. This iterative process can be valuable in other domains where intricate ADDs affect system quality.

RQ$_{4.2}$: *What types of metrics can be applied to evaluate these levels of support, and how effective are they?* In response to **RQ$_{4.2}$**, we introduced generic, technology-agnostic metrics closely aligned with typical ADDs related to automation in MLOps. We programmatically calculated these metrics via our detectors, and ordinal regression analysis statistically reinforced their effectiveness in evaluating automation support. Developing technology-agnostic metrics is advantageous because it enables broad application across diverse MLOps systems. The metrics' effectiveness, as demonstrated by ordinal regression analysis, suggests a promising avenue for objectively assessing automation support.

RQ$_{4.2}$ logically follows from **RQ$_{4.1}$** as it builds upon the foundational work established in addressing **RQ$_{4.1}$**. In **RQ$_{4.1}$**, we rigorously evaluated the support for automation in MLOps systems by manually assessing ADDs and their associated decision options. For **RQ$_{4.2}$**, we introduced generic, technology-agnostic metrics tailored explicitly to these ADDs, expanding upon the evaluation framework developed in **RQ$_{4.1}$** to provide a broader, more generalisable assessment of automation support across diverse MLOps systems, thereby demonstrating a clear relationship between the two research questions and their measurement methodologies.

Our study employs an iterative evaluation process and technology-agnostic metrics to describe a functional modelling and evaluation process for MLOps researchers and practitioners, and our approach includes the modelling of MLOps systems, enabling the objective assessment and evaluation of support for automation-related ADDs. The introduced metrics advance understanding of how to detect and assess the quality of MLOps systems, with scope for broader applicability.

The objective of this study was to determine the extent to which various ADD options are supported. Suppose a metric's value is 0 (or false, in the case of Boolean metrics). In that case, it indicates that the type of automation associated with this decision option is not applied within the system. Conversely, a non-zero value (or true) signifies the presence of the decision option, but it does not guarantee perfect or even correct implementation.

Our methodology and metrics allow us to assess the degree of coverage of automation-related decision options in terms of how well they are supported, enabling us to measure

our intended goals and consider automation as a measurable system quality. An architect familiar with a given project would be aware of the relevant decision options based on the force impacts described in Section 8.4, which provide insights into the preferred options and their trade-offs.

8.8.2 Practical Application

We have demonstrated our approach's effectiveness in detecting and assessing complex quality aspects of MLOps systems, such as ADDs, and pipeline and component properties, which are challenging to identify manually in source code. Incorporating our approach as automated steps within a CI/CD pipeline, as depicted in Figure 8.4, which is described below, would offer substantial practical benefits.

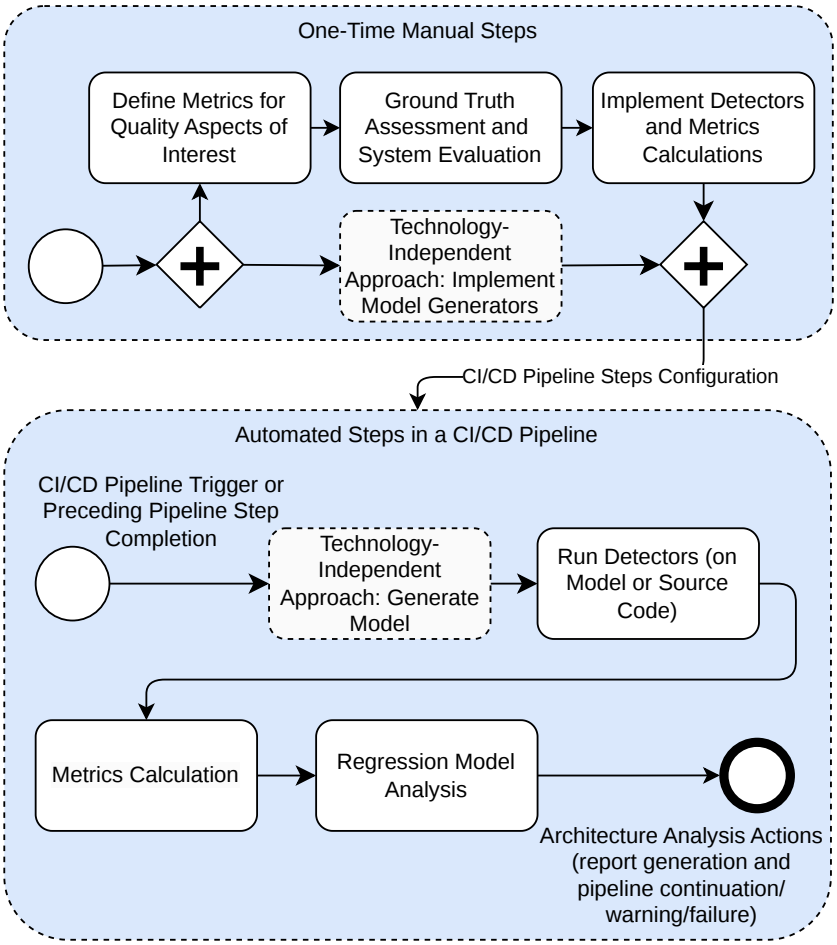

Figure 8.4: A proposed practical application of our solution.

The manual effort required for the initial metrics definition, ground truth assessment, system evaluation, detector implementation and metrics calculations needs to be expended

only once. We demonstrated our approach for this study by manually modelling a range of systems, which required some initial manual effort due to the technology-agnostic nature of our approach. Still, we enabled subsequent analysis automation thanks to the high-level abstraction provided by the models. In a real-world application of a specific system, practitioners can avoid the manual modelling effort by automating the most repetitive and labour-intensive steps for their specific system and its technologies within a CI/CD pipeline, either leveraging source code detectors for a specific set of technologies or automating the generation of system architecture model abstractions for analysis with model detectors if a more technology-independent approach is required.

The subsequent regression model analysis can also be automated within a CI/CD pipeline, generating an outcome report and taking further actions, such as continuing the pipeline, failing or issuing a warning. The proposed application of this solution could be used for regression testing within the quality assurance process to help practitioners automatically identify when quality aspects of their system, in an MLOps context, fall below a predefined threshold. More generally, this automated process can yield early indications of how the system is evolving and whether refactoring may be necessary. Thus, our reusable ADDs, detector-based analysis and regression models facilitate automation, allowing practitioners to invest effort once and reuse results repeatedly. By automating analysis, organisations can gain valuable insights into the qualities of their MLOps systems whilst accelerating the development cycle and ensuring compliance with non-functional requirements, particularly in quality-related areas.

Further extensions could involve more detailed, specific design guidance on measures that could be taken, or automated optimisation of the system architecture with respect to quality attributes, where necessary and appropriate, and depending on the practitioner's needs. Indeed, the scope of application is not limited to automation-related ADDs or automation-related qualities of the system architecture: numerous quality aspects of MLOps systems, specifically those ADDs, qualities and forces that practitioners find most critical on a case-by-case basis, for instance, those that enhance model management and reusability and generally reduce technical debt, could be supported.

8.8.3 Threats to Validity

Ensuring generalisability across diverse MLOps contexts may compromise the external validity of our findings. However, our confidence in the representativeness of our modelled systems remains strong, as they cover a comprehensive set of automation-related ADDs and systems documented in our prior work based on practitioner literature [23], [24].

When modelling MLOps system architectures, there is a risk of unintentionally omitting architectural elements, which can affect external validity. However, our experience in architectural modelling ensured diligent coverage of encountered phenomena, ultimately affirmed by our results.

We studied, modelled and evaluated third-party systems to enhance internal validity and minimise bias in system composition. Despite potential omissions in our search procedures, we mitigated this through a systematic approach, strict inclusion-exclusion criteria and thorough searches. Furthermore, multiple experienced authors rigorously

reviewed the sources.

To maintain internal validity amid potential subjectivity in ground truth assessment, we simplified the evaluation of decision option support. We used straightforward heuristics and a simple three-step ordinal scale. Our goal was to reduce false positives, enable technology-independent evaluation and ensure a non-controversial interpretation of the results.

Although there is potential for interpretive bias in the system modelling process, our experienced author team aimed to develop system models that faithfully represent observed phenomena. However, this risk does not significantly affect our study, as our primary objective was to ensure system model accuracy rather than uniformity with models that might be created by other researchers.

Another potential risk to internal validity involves the authors' development of system variants. However, it is crucial to note that these variants align with documented ADDs and decision options found in the literature [23], [24]. We took care to modify only the necessary aspects for each variant whilst maintaining consistency in all other respects.

8.9 Future Work

Future work involves applying our results to specific industrial systems, showcasing the applicability and practical benefits of our approach in real-world scenarios. We plan to extend our system modelling techniques to cover other types of ML systems, particularly those with less coverage in the scientific literature. Specifically, we wish to apply our approach to RLOps systems, using industry case studies to evaluate support for automation and other core quality aspects through automated, metrics-based analysis using source code detectors and LLMs, rather than the manual modelling step proposed in Section 8.8.2. The solution would involve identifying quality deficits and suggesting necessary extensions by assessing the current coverage of the design space (expressed as ADDs) for MLOps or RLOps systems. We also plan to incorporate our work into an automated process, as suggested in Section 8.8.2, for continuously assessing MLOps or RLOps architectures in an industrial setting. In this setting, metrics are repeatedly evaluated to detect changes in system qualities.

8.10 Conclusion

In this study, we focused on developing a method to assess support for automation-related ADDs and decision options in MLOps systems and demonstrated its feasibility. We followed a systematic process involving sourcing, modelling and assessing various MLOps systems and their variants.

We introduced novel metrics that cover all potential decision options, and we implemented detectors to analyse the modelled systems and compute these metrics. We have demonstrated that simple yet effective metrics provide reliable insights into automation, which is a crucial system aspect that is paramount to practitioners. We employed statistical methods, specifically ordinal regression, to develop prediction models that

demonstrated high accuracy in predicting the ground truth assessment. The validation of our approach and novel metrics rounds off our contribution.

9 Rule-Based Assessment of Reinforcement Learning Practices Using Large Language Models

In the rapidly evolving field of AI, RL plays a pivotal role in developing agents capable of making informed decisions. As these systems become increasingly complex, the need for standardised and automated training methods becomes apparent. This chapter presents a rule-based framework that integrates LLMs and heuristic-based code detectors to assess compliance with best practices in RL training pipelines. We define a set of architectural rules that target best practices in key areas of RL-based architectures, including checkpoints, hyperparameter tuning and agent configuration. We validate our approach through a large-scale industrial case study and 10 open-source projects. The results show that LLM-based detectors generally outperform heuristic-based detectors, especially when handling more complex code patterns. This approach effectively identifies best practices with high precision and recall, demonstrating its practical applicability.

9.1 Introduction

In RL systems, an agent's success in achieving its goals relies heavily on the training strategies used [2]. These strategies include balancing exploration and exploitation, shaping rewards, and applying algorithms such as Q-learning [242] or policy gradients for policy optimisation [136], [243]. Practical training methods are crucial for enhancing learning efficiency and overall system performance.

Several standard best practices or patterns are part of these training methods. For example, checkpoints [1], [244] are used to save a model's state at regular intervals during training, allowing for recovery from interruptions and maintaining efficiency. Similarly, hyperparameter tuning [121] involves adjusting parameters like learning rates and batch sizes. Tuning their values can increase performance. Single-agent versus multi-agent training [141], [142], [143] can be considered based on the task's complexity. A single agent is a more straightforward approach, but it has limitations when scaling up. At the same time, with multi-agents, behaviour can be competitive or collaborative, and is more adaptive in dynamic environments.

We developed a rule-based approach that utilises LLMs to address the challenge of evaluating best practices in RL training pipelines. Our approach defines 31 architectural rules to evaluate RL training practices, realises a flow-based software architecture for LLM-based detection and provides LLM-based detection components for each of the 31

rules. Additionally, we compare them with typical heuristic-based code detectors, which primarily focus on straightforward code patterns.

Heuristic-based detectors are adequate for quickly identifying common code structures, such as specific operations for checkpoint implementations or function calls. However, they struggle to identify more intricate or non-standard code structures, or code in polyglot settings where multiple languages or technologies are used. This challenge is where LLM-based detection can be valuable, as it can interpret the broader context and intent behind the code, as well as handle unforeseen code structures.

In complex case studies, combining both approaches can be particularly effective. For instance, heuristic-based code detectors can detect common code patterns, whilst LLMs can provide deeper analysis where needed, particularly when rules are implemented indirectly. Using these detection techniques, our approach can be used to assess whether RL training strategies are aligned with best practices, even in complex or non-standard code structures or polyglot environments.

This chapter is adapted from **P$_8$**, corresponds to **RC$_8$**, and addresses **RQ$_4$** **RP$_4$** via the following research questions:

RQ$_{4.3}$ *How effectively can LLMs assess best practices in RL training pipelines compared to heuristic-based code detectors?*

RQ$_{4.4}$ *What are the key architectural rules required for compliance with the best practices for checkpoints, hyperparameter tuning, training/agent configuration and single vs multi-agent RL in RL-based systems?*

We tested the effectiveness of our approach through a large-scale industrial case study that uses RL in a CPPS for production automation, as well as 10 open-source RL systems. The industrial case study considers several CI/CD pipelines across production facilities. This comprehensive evaluation allowed us to assess the successful implementation of our approach in different RL environments and training workflows.

This chapter is organised as follows: Section 9.2 introduces RL practices, and related work is described in Section 9.3. Our methodology and the specifics of automatic rule checking are discussed in Section 9.4. The case studies are presented in Section 9.5. The validation is presented in Section 9.6. Section 9.7 interprets the results, whilst Section 9.8 addresses potential threats to validity. Section 9.9 discusses future work and concludes the chapter.

9.2 Background on RL Practices

This section provides background on the RL practices that are central to this work, specifically covering *checkpoints*, *hyperparameter tuning* and *training and agent configuration*.

Checkpoints [1], [244] are essential in RL, as they save the agent's learning state at intervals. Checkpoints allow training to resume from specific points, protecting against data loss and enabling adjustments without restarting from scratch. Regular checkpoints

also enable progress tracking and fine-tuning based on intermediate results, which is key to improving the learning process.

Hyperparameter tuning [121] is an important practice in RL. The success of RL algorithms often depends on choosing appropriate hyperparameters, such as learning rates and discount factors. Adjusting these settings correctly can increase the speed and effectiveness of the learning process. Popular methods for tuning include grid search, random search, and more sophisticated approaches, such as Bayesian optimisation.

A key distinction in RL is between *single-agent* [141] and *multi-agent* [142], [143] systems. Single-agent setups involve one agent learning to make the best decisions by interacting with its environment. In contrast, multi-agent systems comprise several agents that may work together or compete with one another, thereby adding layers of complexity to the learning process. MARL faces additional challenges, such as non-stationarity, where the environment evolves as other agents learn and take action, requiring more sophisticated strategies.

9.3 Related Work

In this section, we discuss studies focusing on RL practices and rule-based detection methods, and compare them with our work.

Many studies have focused on and addressed different aspects of RL methodologies. For instance, Lee et al. [117] investigate the evolution of RL algorithms, highlighting the transition from single-agent systems to multi-agent configurations that rely on distributed optimisation. Canese et al. [118] analyse multi-agent algorithms, emphasising crucial elements such as scalability, non-stationarity and observability, all of which are vital in multi-agent RL contexts.

Eimer et al. [121] provide an in-depth analysis of hyperparameter tuning approaches, covering techniques like grid search, random search and Bayesian optimisation. Li et al. [245] propose the Hyperband method for hyperparameter optimisation, which efficiently reduces computational demand whilst identifying optimal parameters. Our work builds on these studies by incorporating automated hyperparameter tuning detection in RL using a rule-based assessment that leverages LLMs and heuristics.

Chen et al. [244] explore the use of checkpoints in training processes, presenting methods to improve model performance by periodically saving and validating models. Similarly, Eisenman et al. [51] explore checkpointing systems in training recommendation models, outlining how checkpoints can be utilised to boost training efficiency. Our approach aligns with these studies by emphasising the importance of checkpoints, but it also automates the checkpoint validation process directly in the source code, using LLM-based and heuristic-based analysis of specific architectural rules.

Hernandez-Leal et al. [246] present a comprehensive review of multi-agent deep RL, addressing the challenges of non-stationarity, scalability and agent cooperation. Our work also addresses RL challenges, but formulates rules for multi-agent training and validates them using LLM-based and heuristic-based methods.

Zhang et al. [143] provide an extensive overview of MARL algorithms, focusing on

cooperative and competitive dynamics and highlighting the need for robust training practices. Our work complements these insights by implementing rules for multi-agent systems and validating their use in real-world systems.

Schneider et al. [247] present a rule-based approach for verifying security compliance in microservice architectures. Whilst their work centres on security, the rule-based detection system they describe shares similarities with our approach. However, our approach combines rule-based detection with LLM-based and heuristic-based methods to help ensure RL systems adhere to best practices.

9.4 Approach

This section describes the research methods used in this study and our rule-based assessment approach, which leverages LLMs and heuristics to assess conformance to best practices in RL training code. The data used in and produced as part of this study has been made available online in our replication package [248].

9.4.1 Research Methods

Figure 9.1 outlines the research methodology employed in this study.

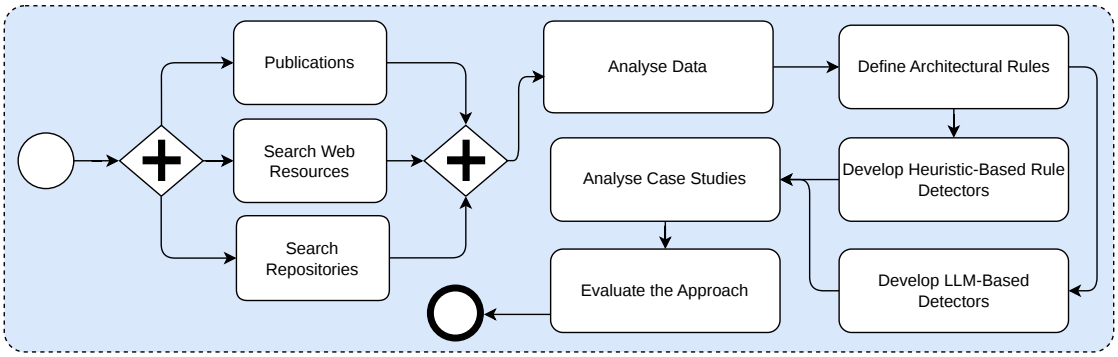

Figure 9.1: Overview of the research method followed in this study.

Publications, Search Web Resources, Search Repositories We initially reviewed various knowledge sources on RL-specific best practices, including publications like books and research papers [1], [121], [123], [134], Web resources and open-source repositories.

Analyse Data Subsequently, we conducted a qualitative analysis using Grounded Theory coding methods [113], such as open and axial coding, to analyse the collected data and extract relevant practices (described in Section 9.2). We closely followed established pattern catalogues, such as those by Lakshmanan [1], for comprehensive insights into RL-relevant concerns. We then, based on this analysis, formulated 31 architectural rules

and developed the corresponding heuristic-based detectors and LLM-based detectors in the steps described below.

Step 1: Define Architectural Rules In the first step, we identified key aspects of RL training that focus on checkpoints, hyperparameter tuning, training and agent configuration. Our approach established specific rules to ensure consistent and correct application of best practices in RL training code. For example, rules were established for saving model states at regular intervals, along with metadata for integrity checks. Hyperparameter tuning rules were defined to validate and adjust parameters essential for performance. Furthermore, additional rules focus on training configurations, managing multiple agents to ensure compatibility, and clearly defining state and action spaces.

Step 2: Develop Heuristic-Based Code Detectors In this step, we designed a set of heuristic-based code detectors that strictly comply with the rules established during Step 1. These detectors parse the RL training code and verify its compliance with the architectural rules. Every code detector is created to recognise practices such as checkpoints, hyperparameter tuning and agent configurations. The detectors function within a service that combines the outputs of these automatic detections to create structured reports in JSON [249] format.

Step 3: Develop LLM-Based Detectors We developed a framework that uses LLMs to handle specifications within the RL training pipeline. This framework enables users to define queries without writing extensive code, as the LLMs interpret these descriptions and generate corresponding answers. By utilising LLMs, the framework can adapt to project-specific requirements and complex representations of RL training practices. The LLM-based detection system can be used in conjunction with the heuristic-based code detectors. This approach simplifies the process for non-expert users whilst maintaining rigorous validation standards.

Steps 4 and 5: Analyse Case Studies and Evaluate the Approach The final step of our approach involved applying the heuristic-based and LLM-based code detectors to several RL systems for evaluation. This process included extensive validation across one industrial case study and 10 open-source RL systems. We analysed the systems using the detectors and evaluated our approach by measuring precision, recall and F_1 scores.

9.4.2 Architecture of the Heuristic-Based Code Detection Approach

Figure 9.2 visually represents the heuristic-based architecture. The `Heuristic Code-Based Detector` comprises several key modules that help validate RL training practices. A central service acts as an `Orchestrator`, processing the source code and managing the analysis workflow to ensure that all necessary validation tasks are executed. `Heuristic-Based Detectors` assess the RL code against best practices, identifying general patterns and common issues. The `Code Analyser` parses the source code, applies these detectors and

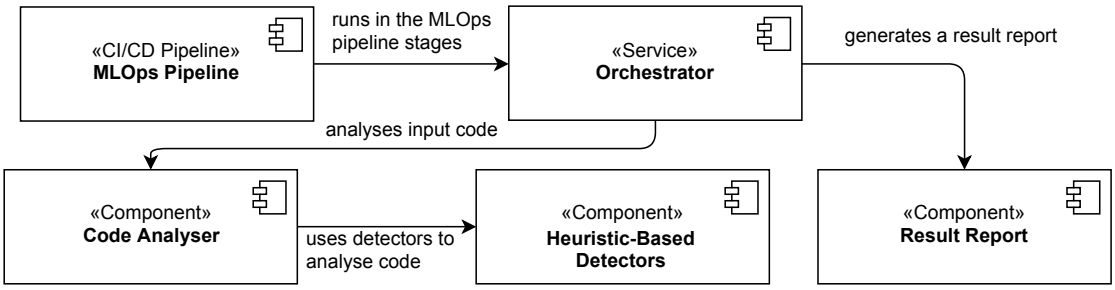

Figure 9.2: Architecture of the heuristic-based detector.

evaluates the code's compliance with best practices, identifying any rule violations. Once the analysis is complete, the `Result Report` component compiles the findings into JSON reports. This service functions within an `MLOps Pipeline` or as a standalone tool.

9.4.3 Architecture of the LLM-Based Detection Approach

The LLM-based approach automates the detection of best practices in RL-based systems. The architecture shown in Figure 9.3 operates in the same context as the heuristic-based approach: it can be used standalone or as part of an `MLOps Pipeline`.

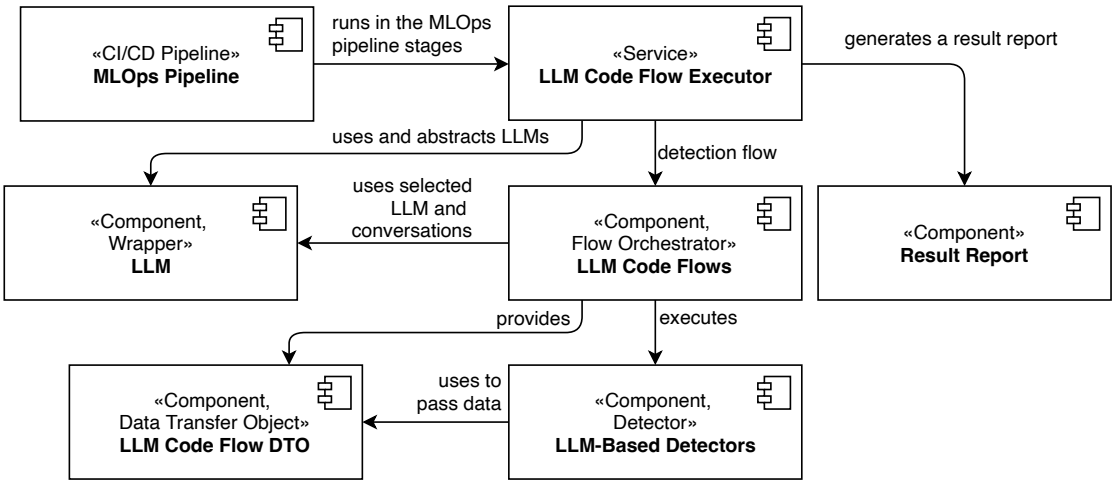

Figure 9.3: Architecture of the LLM-based detector.

The core component, `LLM Code Flow Executor`, manages the execution of the assessment process or a specific validation task. It runs the workflow defined by the `LLM Code Flows` component to execute the required detection and validation steps. `LLM` is a component that abstracts and wraps different supported LLMs. `LLM Code Flow Executor` defines LLM parameters such as model name and token limits, enabling the use of various models, including those from OpenAI, such as GPT-3.5 Turbo and GPT-4o and free models found on Ollama. It also generates a detailed `Result Report`. `LLM Code Flows`

keep track of ongoing conversations, i.e. the interactions with the LLMs, guiding the code detectors based on the context.

The `LLM-Based Detectors` component handles all detection tasks and contains classes that implement the different rules described in this chapter, such as those for checkpoints and hyperparameter tuning. The `LLM Code Flows` provide a data transfer object, `LLM Code Flow DTO`, to each executed detector in the workflow. It contains essential information, including project metadata, source directories and validation status reports. As this object passes through various filters (i.e. different detectors), it generates a report of the detected rules incrementally.

9.4.4 Rules and Detectors

Rules Table 9.1 outlines the rules defined in this study. Each rule is Boolean, meaning it can either be true or false. A rule is true if the corresponding architectural guideline is observed in the application's source code. Conversely, a rule is false if the source code contradicts it or lacks relevant evidence.

Table 9.1: Architectural Rules for RL System Architectures

Rule	Description
Rules on Checkpoints	
R01	Maintain a single, dedicated directory for storing all checkpoints.
R02	Implement adjustable checkpoint intervals to accommodate varying training durations and system performance.
R03	Each checkpoint must include detailed metadata, such as environment specifics, training steps and a timestamp.
R04	Validate the integrity and completeness of each checkpoint post-save using checksums or similar methods.
R05	Ensure that checkpoints are fully compatible with the training policies used, including versions and configurations.
R06	Maintain a history of checkpoints based on configurable retention policies.
R07	Implement mechanisms to compress or selectively store only significant differences between consecutive checkpoints.
R08	Regularly test the recoverability of checkpoints in a controlled environment.
R09	Automate the cleanup of outdated checkpoints based on retention policies.
R10	Checkpoints are created without considering the dependencies or the state of the training environment.
Rules on Hyperparameter Tuning	

Continued on next page...

Rule	Description
R11	Define all hyperparameters clearly and categorise them based on functionality and model components.
R12	Validate hyperparameters before usage to ensure they fall within reasonable and effective ranges.
R13	Employ evolutionary algorithms to explore the parameter space systematically.
R14	Tune hyperparameters considering their interdependencies.
R15	Implement automated tuning systems like grid search, random search or Bayesian optimisation.
R16	Integrate continuous hyperparameter tuning within the training process.
R17	Monitor the impact of hyperparameter changes on model performance continuously.
R18	Apply version control practices to hyperparameter tuning experiments.
R19	Optimise resource allocation during hyperparameter tuning.
R20	Conduct sensitivity analyses to identify impactful hyperparameters.
Rules on Training and Agent Configuration	
R21	Maintain a clear distinction between training parameters and hyperparameters.
R22	Include configurable training options like total timesteps and epochs.
R23	Use consistent seeding across all random number generators.
R24	Track key performance metrics like rewards, losses and evaluation scores.
R25	Configure individual policies for each agent based on specific roles or tasks.
R26	Ensure compatibility across all agents and their policies.
R27	Define consistent observation/action spaces for all agents.
R28	Track performance metrics for each agent individually.
R29	Assign unique identifiers to each agent.
R30	Define a shared interactive environment for all agents.
R31	Clearly define the state and action spaces for each agent.

In the example shown in Listing 9.1, the RL code for creating a checkpoint adheres to rule *R02: Implement adjustable checkpoint intervals to accommodate varying training durations and system performance*, as this parameter is directly used within the *CheckpointCallback* function. In contrast, the architectural rule *R11: Define all hyperparameters clearly and categorise them based on functionality and model components* does not hold, as there are no indications of the defining or grouping of hyperparameters.

Heuristic-Based Code Detectors The detectors were implemented in Python using regular expressions and pattern matching to identify specific code structures. They consist of functions designed to analyse code and verify compliance with architectural rules. Using regular expressions and parsing methods, they identify patterns such as agent configuration and environment management. For example, in RL code that creates checkpoints (see Listing 9.1), heuristic-based detectors look for keywords like `checkpoint` or `.save()` to identify general patterns, or more concrete keywords to identify saving frequency (`save_frequency`) and saving directory (`checkpoint_directory`). However, they may mistakenly capture unrelated functions.

```
from stable_baselines3.common.callbacks import CheckpointCallback

checkpoint_directory = "checkpoints"
os.makedirs(checkpoint_directory, exist_ok=True)

checkpoint_callback = CheckpointCallback(
    save_frequency=1000, # Save the model every 1000 steps
    save_path=checkpoint_directory,
    name_prefix=env.unwrapped.metadata.get('name'))
model.save(f"{env.unwrapped.metadata.get('name')}_
                            {time.strftime('%Y%m%d-%H%M%S')}")
```

Listing 9.1: Source code example of checkpoint implementation (taken from CS1).

LLM-Based Detectors LLM-based detectors use LLMs to inspect source code, spot patterns and check if specific rules are followed. Unlike traditional rule-based methods, these detectors interpret code more flexibly, enabling the identification of patterns that do not strictly adhere to predefined rules. Our LLM-based detectors are highly modular, meaning each checks for specific code fragments that realise part of a rule rather than checking multiple aspects simultaneously. This modularity helps, firstly, to achieve precise, relatively predictable results from the LLMs and, secondly, to establish traceability between the rule and the source-code locations where the code fragments that realise the rule occur.

For example, when dealing with checkpoints, the LLM is instructed to examine the code for `save()` function calls or keywords such as `checkpoint` and then determine whether they align with the specified rules. In the RL code shown in Listing 9.1, the LLM would recognise key parts of the code, such as `model.save()` calls and assess if they match rule R02 by considering how checkpoint intervals are managed.

LLM-based detectors provide more detailed analysis when more nuanced detection is required, such as when verifying that checkpoints are saved in the correct folder. They can track the creation of directories and connect it to checkpoint storage, as shown in the custom detection example in Listing 9.1. This analysis ensures that even complex rules are followed correctly.

The detection process follows several steps: first, the LLM-based approach checks the code for possible issues; then, it compares what it finds with the set architectural rules; and finally, it organises the results in a clear format. This method provides a

comprehensive code review, offering insights into rule compliance and improving validation accuracy. Please note that the LLM is only one of many components required to realise these steps. Figure 9.4 shows a conversation template that establishes a process for interacting with an LLM to review the Python code of an RL project, specifically focusing on the presence of checkpoint functionalities[1].

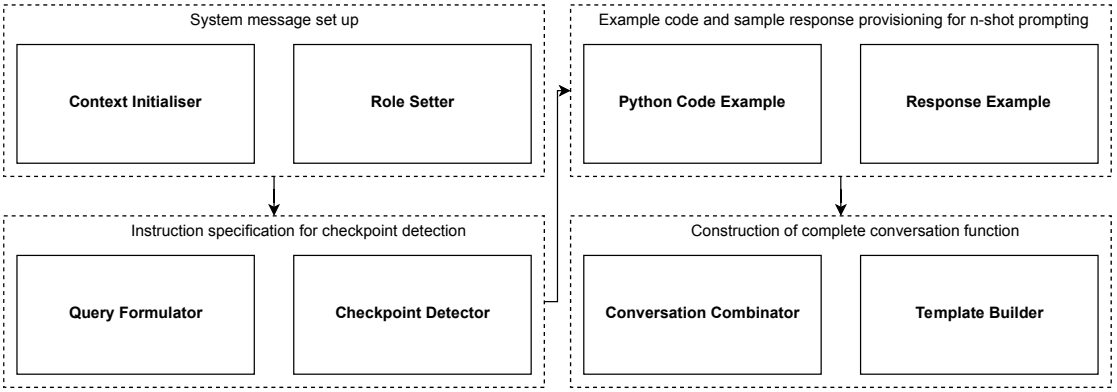

Figure 9.4: The conversation template of the LLM-based detector for checkpoints.

The conversation begins by setting up a system message that instructs the assistant to assist a junior developer in analysing RL code. The assistant is tasked with searching the provided Python file for checkpoint-related code and providing instructions on how to respond if such code is found. The script includes an example of Python code demonstrating checkpoint functionality, along with a sample response that highlights the relevant sections. The corresponding function constructs the conversation by combining the system message, a checkpoint-related query, the example code and response and the target code to be analysed. This structured approach, with clear instructions and examples, helps improve the accuracy and relevance of the LLM's analysis of checkpoint implementations in the target Python code.

9.5 Case Studies

In this section, we describe the case studies used to evaluate our approach and test the performance of the rule detection. We considered one large-scale industry case study and 10 open-source RL-based systems. The case studies are summarised in Table 9.2.

9.5.1 Industrial Case Study

To validate the effectiveness of our rule-based assessment approach, we integrated it into our *ML pipeline insights service* [251] (which is not part of this doctoral thesis) as part of a comprehensive industrial case study. This case study centres on a sophisticated single-agent RL and MARL framework that enables extensive testing and optimisation of

[1] See the detailed conversations and prompts in our replication package [248].

Table 9.2: Overview of Python-Based Systems and their Functionalities

ID	Description
ICS	This system focuses on an advanced framework for single-agent RL and MARL. It enables thorough testing and optimisation of AI policies in diverse environments (See Section 9.5.1 for details).
CS1	In this system, zombies originate at the top border of the screen and move downward along varying, unpredictable trajectories until they reach the bottom border. `https://pettingzoo.farama.org/tutorials/sb3/kaz`
CS2	In this system, Waterworld, the simulation revolves around archaea organisms navigating their environment in a quest for survival. `https://pettingzoo.farama.org/tutorials/sb3/waterworld`
CS3	This system is a chess example. It utilises observation and action spaces similar to those of AlphaZero [250], with slight modifications. `https://pettingzoo.farama.org/tutorials/sb3/connect_four`
CS4	This system is a physics-based game in which the objective is to guide a ball to the left wall of the game's boundary. `https://pettingzoo.farama.org/tutorials/rllib/pistonball`
CS5	This system is a game in which two players must connect four of their tokens vertically, horizontally or diagonally. `https://docs.agilerl.com/en/latest/tutorials/pettingzoo/dqn.html`
CS6	This system is a classic Atari game in which two players control ships, each trying to maximise their score. `https://pettingzoo.farama.org/tutorials/agilerl/MADDPG`
CS7	This system trains AI agents to play noughts and crosses using the PettingZoo environment. `https://github.com/Farama-Foundation/PettingZoo/blob/master/tutorials/Tianshou/3_cli_and_logging.py`
CS8	In this system, there are two agents: the *speaker* and the *listener*. The *speaker* agent possesses the ability to communicate verbally but cannot move autonomously. `https://docs.agilerl.com/en/latest/tutorials/pettingzoo/matd3.html#matd3-tutorial`
CS9	This system is also a classic Atari game, similar to CS6. `https://github.com/vwxyzjn/cleanrl/blob/master/cleanrl/ppo_pettingzoo_ma_atari.py`
CS10	This system is similar to CS1. It trains agents in the "Knights-Archers-Zombies" environment using the Black Death wrapper to handle agent deaths effectively. `https://pettingzoo.farama.org/tutorials/sb3/kaz`

AI policies across diverse environments. The industrial system, developed by Siemens Aktiengesellschaft Österreich, a global provider of production automation solutions, is an advanced platform that features several key components:

- **Agent Management Modules**: these modules enable efficient simulation of agent behaviours and learning processes, allowing for the detailed study of multi-agent interactions and dynamics.
- **Customisable Training Environments**: the system's tools enable precise adjustments to training scenarios, creating varied learning environments tailored to specific strategic tasks.
- **Centralised and Decentralised Learning Approaches**: the platform supports centralised and decentralised learning, optimising training effectiveness through adaptable policy definitions and performance tuning.

- **Hyperparameter Tuning and Optimisation Strategies**: the system includes advanced capabilities for hyperparameter tuning and employs sophisticated optimisation strategies crucial for refining AI behaviours and achieving peak performance.
- **Production Automation Application**: the MARL framework is used for automating production processes, with components such as sophisticated simulators, AI optimisers and training and inference modules.

The heuristic-based detector is integrated into the RLOps pipelines of this system, leveraging a service-based architecture to provide real-time feedback on training and inference. This integration ensures that the training methods adhere to best practices, thereby improving the reliability and performance of AI agents.

The LLM-based detector is also embedded within the RLOps pipelines of this system, using a framework to assess RL training practices. This setup enables more nuanced detection of best practices by interpreting complex code patterns and natural language specifications.

The platform was created by a diverse team of experts and is intended for use on factory shop floors. It aims to standardise and improve production processes by leveraging an advanced MARL framework to address various operational challenges, ultimately boosting efficiency. For this reason, ML specialists and software architects must have a standardised approach to gaining insights into software architectures across different customer projects, especially within the context of their MLOps pipelines. Through this industrial case study, we aim to demonstrate the real-world application and effectiveness of our rule-based assessment tool.

9.5.2 Open-Source Case Studies

The open-source case studies in Table 9.2 (CS1-CS10) cover various RL algorithms, including PPO [136], Deep Q-Network (DQN) [243] and Multi-Agent Deep Deterministic Policy Gradient (MADDPG) [252]. These systems vary widely in scope, from multi-agent platforms for different types of experiments to advanced PPO setups with complex neural networks. Whilst the open-source projects used in this study represent a range of RL algorithms, they do not fully capture the complexity of polyglot or highly complex industrial systems. Each system has a specific focus, such as autonomous play for competitive games or tasks within defined environments. Some systems are designed to be flexible, enabling dynamic interactions across various settings. Many of these systems also aim to enhance decision-making by employing techniques such as action masks in PPO, which help manage restricted or undesirable actions in specific situations.

9.6 Validation

In this section, we validate our approach by demonstrating the applicability and effectiveness of the defined rules in ensuring best practices in RL systems. Our validation utilises a larger industrial case study and 10 open-source case studies of RL systems, as introduced in Section 9.5. This comprehensive evaluation allows us to assess the practical implementation of our rules across diverse RL environments and training practices.

9.6.1 Validation Setup

The validation ground truth was established by manually inspecting the source code for each case study and noting the presence or absence of architectural rules. This manual inspection was conducted by the authors and cross-verified by RL experts to ensure accuracy.

The validation was conducted by applying our two kinds of detectors across multiple case studies. The source code of each RL system was analysed, and the detectors were used to verify whether the code adhered to the defined rules (see Table 9.1). This setup enabled a comprehensive and accurate validation of our approach, utilising both heuristic-based and LLM-based detection methods.

GPT-3.5 Turbo version 0613 was selected to validate our LLM-based detectors due to its robust performance, making it suitable for analysing complex code patterns in RL training pipelines. It balances computational efficiency and performance, offering high accuracy and flexibility without the significantly higher costs associated with larger models such as GPT-4 or GPT-4o. In addition, GPT-3.5 Turbo performs similarly to the best non-commercial LLMs in benchmarks[2]. These factors are essential to consider, as many organisations do not want to or cannot use cloud-hosted, commercial LLMs.

We employed standard evaluation metrics to measure the accuracy of our approach. For measuring accuracy, true positives (TPs) refer to instances where the rule-checking mechanism correctly identifies a source code instance as compliant with the best RL training practices. False positives (FPs) are instances where a source code instance is identified as compliant when it is not. False negatives (FNs) are instances in which the method fails to detect a compliant code instance. True negatives (TNs) are not included in our validation, as our ground truth inspection only marks up true positives, not true negative code instances. We use the metrics defined by Equations 9.1, 9.2 and 9.3 to measure accuracy:

$$\text{Precision} = \frac{TP}{TP + FP} \tag{9.1}$$

$$\text{Recall} = \frac{TP}{TP + FN} \tag{9.2}$$

$$F_1 = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{9.3}$$

9.6.2 Results

In the remainder of this section, we present the validation results assessing the accuracy of the rule-based detectors.

[2] See e.g. `https://www.trustbit.tech/en/llm-leaderboard-juli-2024` and `https://klu.ai/llm-leaderboard`

Industrial Case Study Results

In Table 9.3, we observe the performance of both detector types across the architecture rules. For checkpoint rules, the LLM-based detectors identified seven true positives and missed only one, achieving a precision of 1.0, a recall of 0.875 and an F_1 score of 0.933, indicating high effectiveness in detecting checkpoints. In contrast, the heuristic-based detectors detected three true positives and missed five, yielding a precision of 1.0, a recall of 0.375 and an F_1 score of 0.545, showing reliable precision but lower recall, as some instances were not captured.

For hyperparameter tuning, the LLM-based detectors yielded strong results, achieving five true positives and one false negative, for a precision of 1.0, a recall of 0.833 and an F_1 score of 0.909, indicating high precision and sensitivity. The heuristic-based detectors identified four true positives with two false negatives, yielding a precision of 1.0, a recall of 0.667 and an F_1 score of 0.8, demonstrating good accuracy but a slightly lower ability to detect all instances.

For training and agent configuration, the LLM-based detectors achieved five true positives and one false negative, yielding a precision of 1.0, a recall of 0.833 and an F_1 score of 0.909, indicating good performance. On the other hand, the heuristic-based detectors detected three true positives and three false negatives, resulting in a precision of 1.0, a recall of 0.5 and an F_1 score of 0.667, indicating precise detections but limited sensitivity.

Table 9.3: Precision, Recall and F_1 Scores for the Industrial Case Study

Rules on Checkpoints	True Positives	False Negatives	Precision	Recall	F_1 Score
Heuristic-Based Rule-Based Detectors	3.0	5.0	1.0	0.375	0.545
LLM-Based Rule-Based Detectors	7.0	1.0	1.0	0.875	0.933
Rules on Hyperparameter Tuning	True Positives	False Negatives	Precision	Recall	F_1 Score
Heuristic-Based Rule-Based Detectors	4.0	2.0	1.0	0.667	0.8
LLM-Based Rule-Based Detectors	5.0	1.0	1.0	0.833	0.909
Rules on Training and Agent Configuration	True Positives	False Negatives	Precision	Recall	F_1 Score
Heuristic-Based Rule-Based Detectors	3.0	3.0	1.0	0.5	0.667
LLM-Based Rule-Based Detectors	5.0	1.0	1.0	0.833	0.909

Open-Source Case Studies Results

Across all case studies (see Table 9.4) for checkpoint rules, both heuristic-based detectors and LLM-based detectors showed strong performance in several instances, with high F_1 scores indicating a balance between precision and recall. Notably, the LLM-based detectors achieved perfect F_1 scores of 1.0 in case studies CS4, CS5, CS6 and CS8 for the LLM-based detectors and the heuristic-based detectors for CS1, demonstrating robust detection accuracy.

For hyperparameter tuning (see Table 9.5), the heuristic-based detectors showed variable performance across the case studies. For example, for CS1, CS2, CS3, CS5, CS6,

Table 9.4: Precision, Recall and F_1 Scores for Heuristic-Based and LLM-Based Detectors on Checkpoints

Case Study	Heuristic-Based Detectors			LLM-Based Detectors		
	Precision	Recall	F_1 Score	Precision	Recall	F_1 Score
Rules on Checkpoints						
CS1	1.0	1.0	1.0	0.75	1.0	0.857
CS2	1.0	0.5	0.667	1.0	0.5	0.667
CS3	1.0	0.5	0.667	1.0	0.5	0.667
CS4	1.0	0.5	0.667	1.0	1.0	1.0
CS5	1.0	0.25	0.4	1.0	1.0	1.0
CS6	1.0	0.5	0.667	1.0	1.0	1.0
CS7	1.0	0.5	0.667	1.0	0.5	0.667
CS8	1.0	0.5	0.667	1.0	1.0	1.0
CS9	1.0	0.143	0.25	1.0	0.714	0.833
CS10	1.0	0.667	0.8	1.0	0.333	0.5

CS8, CS9 and CS10, they achieved F_1 scores between 0.667 and 1.0, indicating effective detection in those instances. However, for CS4 and CS7, the heuristic-based detectors struggled, with lower F_1 scores of 0.571, reflecting their limitations in consistently identifying true positives.

Table 9.5: Precision, Recall and F_1 Scores for Heuristic-Based and LLM-Based Detectors on Hyperparameter Tuning

Case Study	Heuristic-Based Detectors			LLM-Based Detectors		
	Precision	Recall	F_1 Score	Precision	Recall	F_1 Score
Rules on Hyperparameter Tuning						
CS1	1.0	0.5	0.667	1.0	0.5	0.667
CS2	1.0	1.0	1.0	1.0	1.0	1.0
CS3	1.0	1.0	1.0	1.0	1.0	1.0
CS4	1.0	0.4	0.571	1.0	0.8	0.889
CS5	1.0	0.5	0.667	1.0	0.667	0.8
CS6	1.0	0.6	0.75	1.0	0.6	0.75
CS7	1.0	0.4	0.571	1.0	0.8	0.889
CS8	1.0	0.5	0.667	1.0	0.667	0.8
CS9	1.0	0.5	0.667	1.0	0.75	0.857
CS10	1.0	0.5	0.667	1.0	1.0	1.0

The LLM-based detectors, on the other hand, generally demonstrated stronger performance, achieving perfect F_1 scores of 1.0 for CS2, CS3 and CS10, and high scores between 0.667 and 0.889 for CS1, CS4, CS5, CS6, CS7, CS8 and CS9, suggesting reliable detection across most cases.

For training and agent configuration (see Table 9.6), heuristic-based detectors displayed varying levels of effectiveness across the case studies. For CS1, they recorded relatively low F_1 scores of 0.286, primarily due to a high number of false negatives. Even when performance improved in cases like CS7 and CS8, the heuristic-based detectors achieved F_1 scores of only 0.727, indicating moderate effectiveness, with a slightly better F_1

score of 0.75 for CS10.

Table 9.6: Precision, Recall and F_1 Scores for Heuristic-Based and LLM-Based Detectors on Training and Agent Configuration

Case Study	Heuristic-Based Detectors			LLM-Based Detectors		
	Precision	Recall	F_1 Score	Precision	Recall	F_1 Score
Rules on Training and Agent Configuration						
CS1	1.0	0.167	0.286	1.0	0.5	0.667
CS2	1.0	0.4	0.571	1.0	0.8	0.889
CS3	1.0	0.375	0.545	1.0	0.5	0.667
CS4	1.0	0.5	0.667	1.0	0.75	0.857
CS5	1.0	0.5	0.667	1.0	0.75	0.857
CS6	1.0	0.5	0.667	1.0	0.833	0.909
CS7	1.0	0.571	0.727	1.0	0.857	0.923
CS8	1.0	0.571	0.727	1.0	0.857	0.923
CS9	1.0	0.5	0.667	0.857	0.75	0.8
CS10	1.0	0.6	0.75	1.0	0.833	0.909

On the other hand, LLM-based detectors demonstrated better overall results, achieving higher F_1 scores across several case studies. For instance, CS6, CS7, CS8 and CS10 achieved F_1 scores of 0.909, 0.923, 0.923 and 0.909, respectively, showing a good balance between precision and recall. Furthermore, the LLM-based detectors recorded an F_1 score of 0.889 for CS2 and 0.857 for both CS4 and CS5, highlighting their effective detection capabilities. The lowest scores were 0.667 for both CS1 and CS3.

9.7 Discussion

This section discusses how the research questions were addressed.

RQ$_{4.3}$: *How effectively can LLMs assess best practices in RL training pipelines compared to heuristic-based code detectors?* The findings across the case studies show that LLM-based detectors are generally better than heuristic-based detectors at checking best practices in RL training pipelines.

For checkpoints (see Table 9.4), LLM-based detectors achieved high F_1 scores in several cases, with perfect scores in CS4, CS5, CS6 and CS8 and an F_1 score of 0.933 in the industrial case study. These F_1 scores indicate strong accuracy, as they could detect checkpoints with both high precision and recall, even in more complex scenarios. In comparison, whilst heuristic-based detectors had high precision, they often missed instances, as seen for CS5 and CS9, with F_1 scores of 0.4 and 0.25, respectively, and the industrial case, where they scored 0.545 due to missed true positives.

For hyperparameter tuning, LLM-based detectors consistently performed better, with high F_1 scores, including perfect scores in CS2, CS3 and CS10 and an F_1 score of 0.909 in the industrial study. Heuristic-based detectors yielded mixed results, with lower F_1 scores in some cases and poor performance detecting true positives in scenarios such as CS4 and CS7, highlighting their limitations in handling more complex patterns.

For training and agent configuration (see Table 9.6), LLM-based detectors again showed higher performance, with strong F_1 scores in cases like CS6, CS7, CS8 and CS10, and an F_1 score of 0.909 in the industrial study, compared to 0.66 for heuristic-based detectors. These values suggest that LLMs can handle more detailed rules during training, whilst maintaining good precision and recall.

As explained earlier, we deliberately report the results for GPT-3.5 Turbo in our validation, as it is currently comparable to the top non-commercial LLMs. We have also performed informal tests with the most recent GPT-4o model, which shows further moderate improvements.

Overall, LLM-based detectors proved more effective, flexible and accurate than heuristic-based detectors for assessing best practices in RL training pipelines. Their ability to handle complex code patterns and varied rules across case studies and industrial contexts makes them a stronger choice for ensuring best practices are met in RL training environments.

Whilst LLM-based detectors have shown significant advantages over heuristic-based detectors in evaluating best practices in RL training pipelines, some notable limitations and considerations must be taken into account. LLMs, especially large models such as GPT-3.5 Turbo and GPT-4o, require significant computational resources, resulting in high operational costs and substantial energy consumption. In many scenarios, heuristic-based detectors may be sufficient, especially for simpler, well-defined tasks. Sometimes, a combination of both approaches makes sense. Our LLM-based detectors are highly modular, with each module designed to check specific occurrences of code fragments. This modularity enables precise, predictable interactions with LLMs, ensuring that each query is focused and directly relates to a specific part of a rule.

RQ$_{4.4}$: *What are the key architectural rules required for compliance with the best practices for checkpoints, hyperparameter tuning, training/agent configuration and single vs multi-agent RL in RL-based systems?* To answer **RQ$_{4.4}$**, we established 31 architectural rules across key areas of RL. These areas included, in particular, checkpoints, hyperparameter tuning, training and agent configuration, including handling single or multi-agent setups.

For checkpoints, we defined 10 rules, including using a dedicated storage directory, setting flexible save intervals, adding details about the training environment and performing checks to ensure data integrity. These help track progress and facilitate recovery of the training process.

Hyperparameter tuning encompasses 10 key rules, emphasising clearly defining and organising parameters, utilising efficient methods to determine optimal values, and monitoring the impact of changes on model performance.

For training and agent configuration, we defined 11 rules focused on separating training and tuning parameters, using consistent settings and tracking each agent's performance to improve repeatability and clarity. These rules also encompass single versus multi-agent configurations, prioritising simpler configurations for single agents, and adaptable strategies for multi-agent systems to handle changes and interactions between agents.

9.8 Threats to Validity

In this section, we discuss the potential threats to validity and outline the steps taken to mitigate them.

Internal validity addresses the accuracy of the results and whether they can be attributed to specific interventions rather than other influences. We ensured internal validity by clearly defining and consistently applying rules with our assessment framework. Detection algorithms were carefully implemented and verified to identify true and false positives. All authors cross-checked results to reduce bias, thus preserving internal validity.

External validity refers to the extent to which our findings can be generalised to other contexts or groups. Our study included a large industrial case and 10 open-source case studies covering different RL systems. To further enhance the generalisability of our results, future work should include testing a wider range of RL systems across multiple domains. Although the diversity of case studies in this validation minimises threats to external validity, expanding the dataset would strengthen the broader applicability of our findings.

Construct validity evaluates whether the study accurately measures the intended concepts. This work is closely linked to the precise specification and accurate detection of best practices in RL training. To mitigate potential threats, we based our rule definitions on a thorough review of relevant literature and established practices. Aware that different interpretations and levels of expertise might affect how rules are defined, we refined them by engaging with RL experts from our industrial partner, Siemens, to ensure they truly reflect industry standards.

A threat concerning LLM-based detectors is the lack of clarity in some instances, which may lead to inaccuracies in identifying RL-related practices. Since these detectors are used on broad datasets, they might not always correctly understand specialised RL systems. Their effectiveness depends heavily on the clarity of the prompts, since ambiguous prompts can lead to missed or incorrect detections. Additionally, applying the prompts to different projects may yield varying results and rule interpretations by the LLMs. Moreover, frequent model updates can introduce inconsistencies, as newer versions may interpret data differently.

9.9 Conclusion and Future Work

This work introduces a method for evaluating best practices in RL training using LLMs and heuristic-based detectors. Key areas of focus include checkpoints, hyperparameter tuning and agent configuration. We applied this approach across various RL systems, covering both an industrial application and open-source projects.

Our findings indicate that LLM-based detectors generally yield better results than heuristic-based ones, particularly when dealing with more complex code patterns. In multiple case studies, the LLM approach achieved greater precision and recall, successfully identifying best practices within RL training pipelines. The modular design of our framework, with numerous supporting components and relatively small modular detectors, is crucial for ensuring precision and traceability in the detection process. This modularity

is also advantageous in settings with multiple RL projects and CI/CD pipelines, enabling ongoing monitoring and improvements. It is essential to consider the limitations associated with LLM-based detectors. These include the high computational cost and energy consumption. Furthermore, whilst LLMs offer superior performance in complex scenarios, heuristic detectors may still be sufficient for simpler tasks, providing a cost-effective and efficient alternative that can also be used in combination with LLM-based detectors.

Future work should build on this approach to accommodate more diverse and complex case studies, refine current rules, and create new ones tailored to different algorithms and frameworks, thereby further validating our approach. Furthermore, we plan to integrate our rule-based evaluation into other CI/CD pipelines, enabling real-time validation of RL practices to enhance the reliability and efficiency of those systems.

Part VI

Pipeline Generation for MLOps and RLOps Using Large Language Models

10 MLOps Pipeline Generation for Reinforcement Learning: A Low-Code Approach Using Large Language Models

MLOps and its application to RL pose various challenges when integrating ML and RL models into production systems, requiring considerable expertise and manual effort that can be error-prone and obstruct scalability and rapid deployment. We propose a new approach to address these challenges in generating MLOps pipelines. We present a low-code, template-based approach leveraging LLMs to automate RL pipeline generation, validation and deployment. In our approach, the Pipes and Filters pattern enables fine-grained generation of MLOps pipeline configuration files. Built-in error detection and correction help maintain high-quality output standards.

To empirically evaluate our solution, we assess the correctness of pipelines generated with seven LLMs for three open-source RL projects. Our initial approach achieved an average error rate of 0.187 across all seven LLMs. OpenAI GPT-4o performed best with an error rate of just 0.09, followed by Qwen2.5 Coder at 0.15. We implemented a single round of improvements to our implementation and low-code template. We reevaluated our solution using the best-performing LLM from the initial evaluation, achieving perfect results with an average error rate of 0 for OpenAI GPT-4o. Our findings indicate that pipelines generated by our approach have low error rates, potentially enabling rapid scaling and deployment of reliable MLOps for RL pipelines, particularly for practitioners lacking advanced software engineering or DevOps skills. Our approach contributes towards demonstrating increased reliability and trustworthiness in LLM-based solutions, despite the uncertainty hitherto associated with LLMs.

10.1 Introduction

MLOps [6] refers to the automated provisioning of ML models through the implementation of repeatable workflows. These workflows help to ensure process reproducibility and can potentially mitigate risks associated with model management and deployment [31], [57]. Despite noteworthy progress in MLOps, critical challenges remain in integrating ML models into production systems. Conventional methods often require considerable manual effort to transition ML scripts from experimental environments, like computational notebooks in Google Colab or Project Jupyter, primarily used by data scientists, to production-ready delivery pipelines [58]. This manual hand-off creates inefficiencies and increases the

likelihood of errors, obstructing scalability and rapid deployment [59]. Consequently, there is a pressing need for methodologies that can automate and simplify the creation of production pipelines for MLOps, reducing reliance on extensive, specialised DevOps and software engineering expertise [31], [60]. The application of MLOps to RL is sometimes called RLOps [7], and although MLOps refers specifically to ML tasks, it equally applies to RL in the form of RLOps [8], [9]. In this chapter, we focus on RL pipelines. However, since RLOps is a specialised version of MLOps and has not yet fully achieved widespread terminological adoption, and the proposed approach can be applied equally to both MLOps and RLOps, we primarily use the term "MLOps" unless the distinction is relevant.

This chapter addresses the challenges described above (manual effort, inefficiencies, error-proneness, scalability, deployment velocity, automation and expertise) by proposing a novel approach to automating MLOps CI/CD pipeline generation. It implements the Pipes and Filters pattern [253], applies few-shot prompting and prompt chaining [254], error detection and retries to guide LLMs in generating GitLab CI/CD pipeline configuration files based on low-code, template-based MLOps CI/CD pipeline specifications for the RL lifecycle.

Our method employs a low-code approach grounded in template-based natural language specifications. A textual generation template is provided that can be adapted with natural language-based specifications to guide the practitioner. Coding knowledge is needed only to review inputs and generated artefacts.

Our approach automatically generates, validates and deploys RL pipelines. It utilises LLMs to generate production-level systems from initial RL scripts and user-specified generation requirements. These LLMs, specialised through prompt engineering techniques and integrated with supporting components, provide modular, flexible automation suited to diverse deployment contexts.

The modular architecture of our approach employs the Pipes and Filters pattern, orchestrating the flow between various LLM-based generation modules and supporting components. Each module specialises in discrete tasks, such as creating Dockerfiles [33], generating CI/CD pipelines or issue detection. The architectural design provides quality assurance in various tasks, including generation retries, linting, running validation tasks and automated error correction, enhancing the reliability of the generated pipelines.

In addition, the natural language–based specification supports this generation architecture, enabling non–software engineering experts, such as data scientists and domain specialists (e.g. in CPPSs [13], [14]), to actively contribute to pipeline development – assuming that supporting infrastructure will be provided to them – and offers an alternative to manually writing pipeline specifications and using LLM code completions, since it obviates the need for specialist knowledge on how to structure specific pipeline configuration implementations. Although a formal user study is planned for future work, early feedback from demonstrations with our industry project partner, Siemens Aktiengesellschaft Österreich, indicates that domain specialists can use our solution to author and refine pipeline descriptions without prior LLM and MLOps expertise to generate reliable GitLab pipeline configuration files. Siemens is a large multinational technology company providing production automation solutions for smart factories. Its employees develop RL-based Industry 4.0 [10] CPPSs, including manually-written RLOps

pipeline configurations, and manage their own infrastructure.

Our solution accommodates many pipeline architectures, from straightforward CI/CD pipelines dedicated to training and evaluation to more detailed Dockerised pipelines that involve comprehensive testing, error handling, artefact management and deployment. Thus, both standard and advanced practices are catered for.

We evaluate three open-source RL examples and compare the capabilities of seven LLMs to assess our approach. Our empirical evaluation compares the performance of our automated pipeline generation with expected pipelines, focusing on error rates and execution success.

This chapter corresponds to **P9** and **RC9**, and addresses **RQ5** and **RP5** with the following research questions:

RQ5.1 *How accurately can a low-code, template-based modular flow approach – leveraging LLMs for generation and validation – produce correct MLOps pipeline configurations for RL?*

RQ5.2 *How accurately do different LLMs generate MLOps pipeline configurations for RL, and what are their limitations?*

Our work makes two significant contributions. Firstly, we implemented a modular flow architecture based on the Pipes and Filters pattern. This flexible yet powerful design means that specific components have well-defined responsibilities, making our approach efficient and precise. Our implementation also provides retry, validation and error-correction features, resulting in highly reliable generated pipelines. Secondly, our low-code, extensible, template-based approach provides guided, convenient, powerful and user-friendly natural language specifications for describing and designing pipeline architecture steps, enabling stakeholders without software engineering or DevOps skills to participate in the development process in a cross-disciplinary fashion. All project artefacts, including the implementation, are available in our Zenodo replication package [255].

The rest of the chapter is organised as follows. In Section 10.2, we describe RL and MLOps CI/CD pipeline architectures, and in Section 10.3, we discuss related studies and compare them with our work. Section 10.4 presents the approach we adopted in this study and provides an illustrative example of the approach in action. Section 10.5 describes our study objects, evaluation scenarios, evaluation variables and the LLMs. Section 10.6 details our initial results, and Section 10.7 describes the improvements we made to our solution and the results of our reevaluation. Section 10.8 describes how our solution can be integrated into an industrial setting, and in Section 10.9, we discuss our results in the context of our research questions. In Section 10.10, we consider potential threats to the validity of our study. Finally, Section 10.11 concludes the chapter and suggests future work.

10.2 Background

This section provides background on ML, RL and tasks associated with the RL workflow when creating MLOps pipelines relevant to this study. We also describe aspects of MLOps and MLOps CI/CD pipeline architecture pertinent to this chapter.

10.2.1 RL

ML and RL [2] use differing approaches to their operational methods. ML uses supervised, semi-supervised or unsupervised learning techniques to extract predictions from data, and RL trains agents for sequential decision-making through environmental interactions and reward-based feedback. The process requires agents to describe their problem, states and actions, and also defines a reward system.

Continuous performance enhancement follows an iterative training process that starts by setting up an environment, defining a policy for model training algorithms and then adjusting hyperparameters and evaluating the policy. Differences between ML and RL warrant acknowledgement during the development of MLOps systems and CI/CD pipelines aimed at automation. For example, RL uses complex model architectures and distribution strategies, such as model checkpoints, which impose requirements on the system architecture and necessitate a checkpoint management system that affects data storage, retrieval and deployment mechanics [8].

Since this study focuses on CI/CD pipelines for RL, and we involve minimal external components for the RL projects we use, the RL details are confined to the source code of the RL scripts executed at various stages of the defined CI/CD pipelines. Hence, we are concerned with a higher level of abstraction than the low-level implementation details – we are more interested in the nature and order of the executed RL pipeline stages. Pipeline architecture is discussed in detail in Section 10.2.2.

10.2.2 MLOps CI/CD Pipeline Architecture

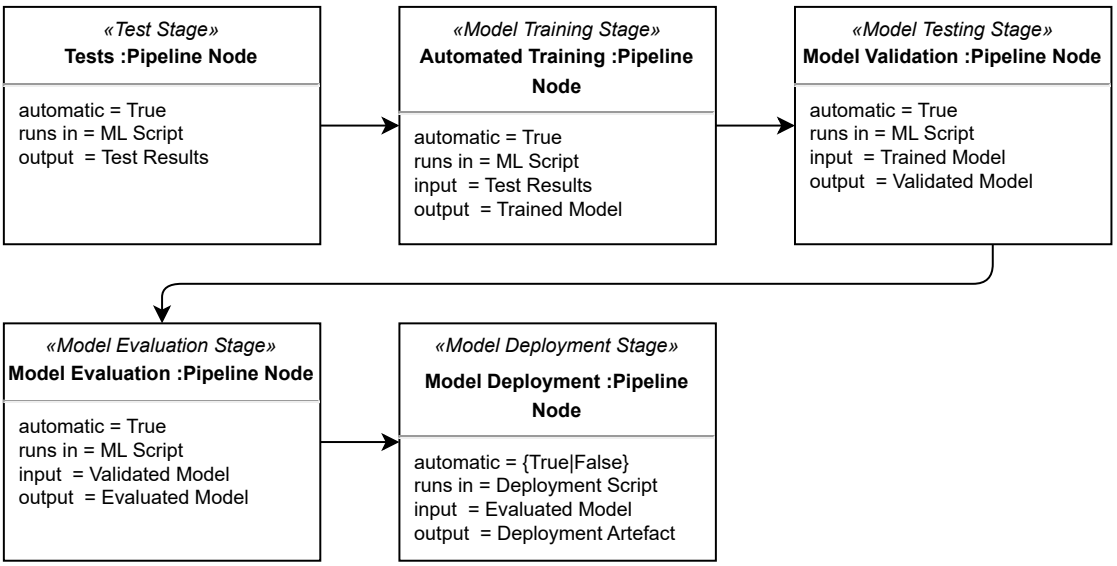

Figure 10.1: A typical MLOps CI/CD pipeline architecture.

MLOps is an all-encompassing approach to the ML workflow [23] that includes model

training, deployment and the management of ML models in production environments [4] and defines processes and practices for model lifecycles. It has much in common with DevOps [21], [28] since it concerns process automation and integration, as well as the use of CI/CD for builds and deployments.

This study focuses solely on CI/CD-relevant aspects of MLOps [22], [165]. CI refers to code changes being tracked, tested and reported to ensure code correctness and provide fast feedback whenever a change breaks the project. CD builds upon CI and involves the automatic or semi-automatic packaging and deployment of software to facilitate frequent, low-friction releases. Other aspects of relevance to CI/CD include software builds, environment provisioning and various testing processes.

We focus on automating the build, training and deployment processes for RL models. Inapplicable ML-specific practices that one would expect to find in ML CI/CD pipelines [23], [24] are excluded because they are irrelevant to RL. Such practices include, for instance, data processing pipeline tasks like data extraction, data transformation and data validation. Limiting the scope in this way means the study focuses on CI/CD pipeline configuration for RL, although the approach can also be applied to ML.

The MLOps CI/CD pipeline architecture comprises well-defined stages, most of which run automatically, accelerating the creation and deployment of ML or RL models. A typical MLOps CI/CD pipeline architecture for RL is depicted in Figure 10.1. It consists of five key stages relevant to this study: *tests*, *model training*, *model testing*, *model evaluation* and *model deployment*.

The *tests* stage focuses on source code. It may invoke various test types, such as unit tests, integration tests, performance tests, security scans and static source code analysis. Unit tests check the implementation for correctness, whereas integration tests verify the interaction between different components. Performance tests evaluate the system's performance metrics, and security scans identify potential vulnerabilities. Static analysis evaluates code for possible problems before the program is executed.

During the *model training* stage, automatic training of the RL model takes place using specified algorithms. This stage also provides a model artefact for subsequent phases.

The *model testing* stage validates the trained model against predefined criteria. It checks whether the model meets these criteria before it proceeds to the *model evaluation* phase.

The model is considered deployment-ready if the *model evaluation* stage confirms its operational performance. Key activities in this stage may include evaluating model performance using suitable metrics and A/B testing to determine which version produces the best results. Stress testing can also reveal a model's abilities during critical conditions. Finally, predictive performance checks may check the model's predictive capabilities.

The final MLOps CI/CD pipeline stage – *model deployment* – entails deploying the model to one or more operational platforms within the associated target environments. A staging environment allows users to perform final testing. After satisfying all the testing criteria and verification requirements, the model can be deployed to a production environment. Deployment to a production environment enables clients to use the model. Blue-green deployment allows a new model version to be deployed alongside existing versions. Another deployment option is canary deployment, in which a new model version

is deployed to a minimal number of clients before finally being fully rolled out. Deployment to a specific environment can be automatic or executed with minimal manual intervention (typically a single click by a human operator within the CI/CD platform).

The pipeline definitions used in this study's open-source projects use only a subset of the functions described above. To focus on the core stages and activities within the pipeline architecture, we also omit many of the details described in Figure 10.1. We exclude external collaborating components such as source code repositories, metadata stores, model registries and feature stores, and deployment environment practices such as environment provisioning, model performance monitoring and drift detection. Furthermore, pipeline triggers, such as triggering a pipeline upon a change to the codebase, are also left out because they are defined externally to the CI/CD pipeline configurations. Finally, some solutions may include pipeline stages or activities that are irrelevant to this study, such as packaging or containerisation of deployment models.

10.3 Related Work

This section describes related work, covering MLOps and RL in a CI/CD context, low-code programming, as well as LLMs for code generation.

10.3.1 Related Work on MLOps and ML in a CI/CD Context

MLOps is widely used because it aims to improve the deployment of ML models through automated processes. Vemuri and Thaneeru [256] investigate the incorporation of AI-optimised DevOps within cloud CI/CD environments, outlining the journey from design to implementation. In prior work [24] in this doctoral thesis, we analysed ADDs for ML model deployment and emphasised the importance of CI/CD automation in such architectures. We also examined architectures designed explicitly for ML workflows [23].

Zhang et al. [257] conducted another study on architectural decisions for AI-based systems and noted that these decisions are primarily technical and closely tied to the specific characteristics of those systems. Kumara et al. [258] present a reference architecture for MLOps, based on a review of industry literature, to address the challenge of selecting appropriate tools and design strategies for developing MLOps environments. This work complements these studies by offering a low-code, template-based approach for automating RL pipeline generation.

Several studies also address the difficulties of incorporating ML into established DevOps practices. For instance, Sharma et al. [259] examine alternatives for implementing ML techniques within CI/CD pipelines but fail to provide a complete solution. Liang et al. [260] focus on automating model training, promoting transparency through version control, and integrating ML into conventional CI/CD pipelines, whilst stressing the importance of ongoing monitoring and feedback loops to sustain model performance. In contrast, our research introduces an innovative framework for automating RL pipeline generation using LLMs and a low-code methodology, aiming at a broader user base, including those without expert knowledge.

10.3.2 Related Work on Low-Code Programming

Interest in low-code development methods has increased considerably in recent years [261], [262]. Hirzel [263] attempts to demystify the term "low code" and discusses how low-code programming reduces reliance on traditional text-based programming languages such as C, Java and Python, presenting low code as a different approach that uses natural and visual languages that resemble user task thinking patterns. Our study makes use of this concept by using a low-code, template-based, natural language approach for pipeline specification, leaving code generation to LLMs.

Kirchhof et al. [264] observe that low-code development platforms (LCDPs) commonly apply one-shot generation. This code generation style creates problems because the code is overwritten during subsequent generation phases, undermining the core concepts of low-code platforms. To reduce this problem, our approach implements few-shot prompting, code validation through error detection and automatic retry functions for generation. Our approach eliminates the need for LCDPs by enabling easy pipeline definition with simple, natural language-based templates.

Bruhin et al. [265] consider the connection between LCDPs and generative AI. Their research incorporates expert interview findings, demonstrating how generative AI can transform the potential of LCDPs. The study indicates that code generation through this combination can boost productivity and minimise errors. Our approach aims to demonstrate the benefits of combining low-code with generative AI without requiring LCDPs. Instead, we opt for a natural language template specification for pipelines.

10.3.3 Related Work on Using LLMs for Code Generation

Several related works aim to understand and enhance LLM outcomes for code generation tasks, ranging from improvements in code generation capabilities to addressing challenges and limitations, to integrating LLMs with other technologies and processes, and to focusing on practical applications.

A systematic assessment of LLM code generation challenges is performed in an analysis by Jin et al. [266]. The six-layer vision model they describe divides code generation operations into four sequential phases: *input*, *orchestration*, *development* and *validation*. Their study seeks to broaden scientific understanding of LLM-based code generation programs to accelerate their development. The authors present specific guidelines and suggestions to enhance the reliability, robustness and usability of LLM-based code generation systems.

The study by Jiang et al. [267] presents a survey of advancements in LLMs for code generation tasks. The study develops a taxonomy for recent developments, covering data curation, model performance evaluation and real-world applications. The authors employ established benchmarks for empirical testing, providing insights into the evolution and effectiveness of LLMs in code generation. Their study also identifies critical challenges and promising opportunities related to the gap between academia and practice and establishes an online repository to document advances in the field continuously.

Li and Murr [268] explore how current GPT-4 models function for generating synthesised programs. They introduce a repository that connects these models to the

HumanEval [269] dataset for zero-shot evaluations of Python code generation. Their study establishes that GPT-4 model variants can effectively solve HumanEval problems and achieve competitive performance with multi-step prompts, demonstrating their potential for enhancing program synthesis.

Li et al. [270] address natural language understanding in code-focused LLMs. Their novel framework comprises two modules: *AttentionExtractor* acquires user requirement statements and *AttentionCoder* generates code from these statements. This framework enables traditional natural language processing tools to integrate with code generation models, enabling them to execute user intents across programming languages.

Our approach complements the above studies by utilising LLMs to improve the flexibility and robustness of MLOps pipeline generation by incorporating natural language processing. LLMs can handle unpredictable or polyglot code [271], allowing them to seamlessly interpret and generate code across multiple programming languages or dialects. This adaptability is crucial when meeting project requirements and integrating with existing systems. Furthermore, LLMs allow users to extend low-code specifications with customised functions or unforeseen extensions, facilitating innovation and customisation without requiring in-depth software engineering knowledge.

Using LLMs for issue detection and code generation introduces accuracy challenges [272], [273], including incorrect classifications and AI hallucinations. The model may generate implausible or irrelevant suggestions due to overfitting, insufficient training data or a limited token window size [274]. White et al. [275] identify issues using LLMs like ChatGPT to generate application programming interface (API) specifications from natural language requirements and simulate APIs to enable developers to interactively test APIs without deploying a mock server. They suggest generator and prompt engineering patterns to expedite the initial API development by generating synthetic data from natural language descriptions, but do not achieve perfect results.

LLM specialisation and fine-tuning, often combined with model evaluation and testing, are essential for achieving acceptable outcomes [276], [277]. LLMs' limitations in accuracy and reliability as coding assistants have been well documented [278]. They can handle simple programming tasks but struggle with detailed descriptions and have limited attention spans, making their performance highly dependent on the specific inputs and prompts used [279]. Whilst LLMs show promise in automatic program repair [280], [281], they often replicate common programming mistakes found in human-crafted solutions [282]. In security code reviews, LLM-generated responses tend to be verbose, vague and incomplete [283]. Given that generating comprehensive CI/CD pipelines involves complex architectural considerations, similar accuracy and reliability issues are expected.

Our approach heeds the lessons of prior work by focusing on the architecture of the supporting components (as also advocated by Tan et al. [276]) and employing a structured generation flow with focused steps and contextual feedback. Our solution enables detailed, natural language, template-based user input to further improve the generation process. Our solution also addresses the reliability and trustworthiness issues inherent to LLMs through built-in quality control and retry logic, validated across two rounds of evaluation. To our knowledge, no similar approach has been suggested in prior work.

10.4 Approach

In this section, we describe the key constituents of our approach: our research process, the traditional approach, the specification template, the proposed workflow, an illustrative example of the template-based low-code approach, the prompting method, the generation architecture, and the definition of generation flows within the implementation.

10.4.1 Research Process

We followed the systematic, structured methodology shown in Figure 10.2. The first step

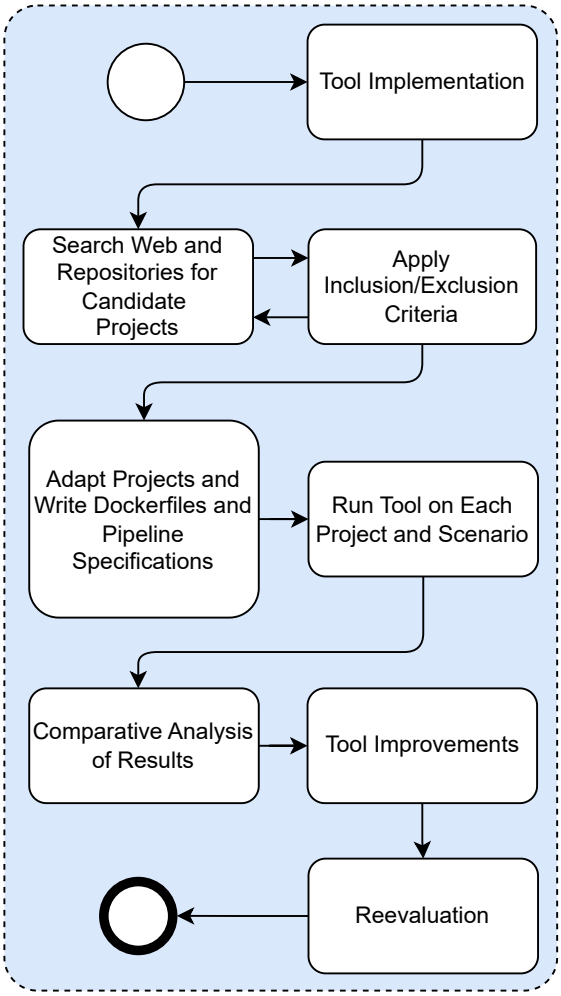

Figure 10.2: The research process followed in this study.

(**Tool Implementation**) was to implement the approach described later in this section. We then searched for open-source RL projects on the Web and in code repositories (**Search Web and Repositories for Candidate Projects**) to find candidates to

evaluate our approach. We iteratively searched and applied specific inclusion/exclusion criteria (described in Section 10.5.1) to determine which projects were suitable for the study (**Apply Inclusion/Exclusion Criteria**). Next, we adapted each project for CI/CD pipeline suitability and wrote Dockerfiles and pipeline specification files for each included project (**Adapt Projects and Write Dockerfiles and Pipeline Specifications**). The pipeline specification files were used as inputs to our implementation, as detailed later in this section, and we also used them to compare against the LLM-generated pipeline configurations during evaluation. Once we had selected appropriate projects (see Section 10.5.1 for details) and prepared the files as described in the previous step, we ran our implemented approach on each project, scenario and LLM three times and recorded the output (**Run Tool on Each Project and Scenario**). We subjected the pipeline configurations generated by the LLM to a comparative analysis against the original pipeline specifications, reference configurations, and the expected behaviour of the executed pipelines, to understand how our approach performed across different projects, scenarios and LLMs (**Comparative Analysis of Results**). We then improved our implementation (**Tool Improvements**) and reevaluated the best-performing LLM from the first evaluation (**Reevaluation**). Section 10.5 describes the scenarios and evaluation variables, Section 10.6 describes the initial results, Section 10.7 highlights the improvements we made and the results of the reevaluation, and Section 10.9 discusses our results. The rest of this section describes our approach and its implementation as a tool.

10.4.2 Traditional Approach

Building and deploying ML models requires complex pipelines that encompass testing, training, validation, evaluation and deployment. Traditionally, this is a highly manual, error-prone process that requires specialised DevOps/MLOps expertise. Errors in these pipelines directly impact the trustworthiness of the resulting ML system. If the pipeline is flawed, the ML model or RL agent will be trained and deployed with potentially incorrect or biased data, leading to unreliable behaviour.

Considering GitLab CI/CD configuration as an example demonstrates the high complexity of the conventional pipeline setup procedure. Manual GitLab CI/CD pipeline setup involves creating and managing `.gitlab-ci.yml` files, which specify the complete pipeline structure in YAML. Practitioners must clearly define each part of the pipeline, including stages, jobs, scripts, dependencies, variables and deployment targets. In MLOps applications, these settings are much more complicated as the testing, model training, model testing (validation), model evaluation, and model deployment stages need to be orchestrated[1].

The setup process usually starts with defining pipeline stages. Every task in these stages must be configured manually with appropriate Docker images, scripts, artefacts and environment variables. The practitioner should also configure GPU[2] runners for training jobs, set data mount points and set model storage locations.

[1] `https://docs.gitlab.com/ci`

[2] `https://blogs.nvidia.com/blog/what-is-mlops`

A standard MLOps pipeline setup in GitLab requires complex job orchestration, requiring ML engineers to liaise with DevOps engineers to establish artefact flows between the training and deployment phases. The configuration needs to support large model files beyond the scope of Git. It thus needs to be integrated with external model registries or model stores, or stored as GitLab artefacts, managed carefully as binary files. Conventional methods require manual definition of resource needs, timeout settings, retry strategies and failure management systems on a per-job basis[3].

The configuration's complexity is further increased by the need to support multiple environments (development, staging and production) with varying resource constraints and security requirements. Every environment needs a different job configuration, variable management and deployment strategy, which often means hundreds of lines of YAML configuration. The user has to manually specify conditional job execution, dependencies between jobs and the correct artefact propagation across the pipeline[4].

The classic GitLab CI/CD setup procedure for MLOps introduces several layers of complexity that even seasoned professionals can find challenging to handle. The YAML syntax is also highly formatted and indented, so configuration files are error-prone and hard to debug. Complex pipeline architectures require advanced YAML features such as anchors, aliases and the `extends` keyword to reduce code duplication, which introduces additional syntactic complexity[5].

The conventional method also requires extensive expertise in GitLab's advanced features, including parent-child pipelines to support complex MLOps processes, multi-project pipelines and conditional pipeline execution based on code changes. These features can be learned only with a considerable investment in GitLab-specific expertise, which may not apply to other CI/CD systems[6] and do not generally fall within the area of knowledge of ML engineers.

Conventional GitLab CI/CD setup methods pose a significant scalability bottleneck when organisations seek to deploy numerous ML models or scale their MLOps capabilities. Every new model or use case also necessitates a new configuration file, duplicating and slightly modifying pipeline definitions. This proliferation of configurations requires high maintenance because updates to common patterns have to be manually applied to many files[7].

Large organisations face specific challenges in standardising MLOps pipelines across multiple teams and projects. Conventional methods lack a robust system for sharing configuration patterns, so each team ends up with different versions of similar pipelines. This disintegration leads to inconsistency in deployment, security and monitoring within the organisation[8].

---

3 https://about.gitlab.com/blog/continuous-machine-learning-development-with-gitlab-ci

4 https://about.gitlab.com/blog/efficient-devsecops-workflows-with-rules-for-conditional-pipelines

5 https://docs.gitlab.com/ci/yaml/yaml_optimization

6 https://about.gitlab.com/blog/parent-child-vs-multi-project-pipelines

7 https://adaptavist.com/blog/top-three-challenges-to-scaling-cicd-in-the-enterprise

8 https://dev.to/zenika/gitlab-ci-10-best-practices-to-avoid-widespread-anti-patterns-2mb5

The conventional GitLab CI/CD setup process introduces significant delays in the pace of development and deployment in MLOps processes. The manual configuration process makes creating new MLOps pipelines take multiple weeks rather than multiple hours when complex ML-specific needs are involved. Every configuration change needs thorough testing and verification to make sure that the pipeline runs correctly, which slows down the iterative development process[9].

Debugging conventional pipeline setups is a major challenge, as YAML syntax errors, dependency conflicts and resource provisioning issues are not always easy to identify or fix. The lack of advanced validation tools means configuration errors are only detected during pipeline execution, leading to failed runs and deployment delays. This iterative configuration debugging and testing process may slow down pipeline development[10].

Change management also slows down as pipeline complexity increases, with traditional configurations requiring extensive manual review and testing for each change. The absence of automated configuration validation also implies that even the slightest alterations may cause pipeline failures, requiring careful change management procedures that slow deployment. Configuration change integration testing across multiple environments introduces additional time overhead to the deployment process [284].

The complexity, expert knowledge requirements, scaling limits and long deployment cycles undermine the reliability of pipeline configuration in multiple ways. The problem with the traditional MLOps pipeline paradigm, which is burdened by complex manual setups, practitioner experience and knowledge bottlenecks, scalability issues and slow release cycles, is that it essentially negates the qualities of transparency, accountability, resilience and reliability that define software trustworthiness.

The complexity of CI/CD systems makes them opaque. With pipelines composed of sprawling, hand-crafted YAML definitions and custom scripts, it becomes almost impossible for anyone but the original authors to understand the system's behaviour. This lack of transparency compromises trustworthiness because stakeholders cannot determine how the pipelines function or whether they function correctly.

10.4.3 Template-Based Low-Code Approach with Natural Language Elements

Our approach leverages template-based natural language specifications to generate the GitLab pipeline. The template provides a structured format that can be augmented with natural language text for interpretation by the LLM. Templates enhance input consistency and reduce errors. Using our templates with LLMs improves the accuracy and dependability of the generated results. It combines the advantages of human-readable formats with the precision of machine-generated outputs, thereby improving reliability.

Table 10.1 describes the main sections of the template. This structured template serves as a detailed guide for creating the RL pipeline. All pipeline details can be specified using

[9] https://about.gitlab.com/blog/build-an-ml-app-pipeline-with-gitlab-model-registry-using-mlflow

[10] https://about.gitlab.com/blog/guide-to-ci-cd-pipelines

Table 10.1: Descriptions of the Low-Code Template's Sections

Section Header	***Description***
`Usage`	An example of how to run the scripts of the RL example. Based on this, the LLM should determine how to run the stages (assuming the scripts are named reasonably and their parameters are specified).
`Required Files`	A list of essential files needed for the project, such as ML, testing and environment setup scripts. These should be used and adapted for the project's training, evaluation, testing and deployment.
`Requirements`	The Python dependencies required for the project.
`Environment Variables`	The environment variables necessary for the execution of the project, along with their intended default values, e.g. to specify how many episodes model training should run for.
`Gitlab CI pipeline stages`	The required stages of the CI/CD pipeline, such as `unit_test`, `train`, `model_test`, `eval` and `deploy`.
`Gitlab CI stage requirements`	Specific requirements for each CI stage, such as artefact generation in the `train` stage and dependencies between stages. Provided as natural language text.
`Gitlab CI pipeline results`	Indicates whether the pipeline generates certain results, such as artefacts or Docker images. The generation of artefacts can lead to substantially different pipelines, e.g. if Docker images, Docker-in-Docker or a Docker-enabled pipeline runner are required.
`Details of the deploy stage`	Information on the deployment process, including the target server, user access credentials and deployment tasks to be executed on the server.

the template. By offering low-code components within a template along with natural language specifications, we can restrict the LLM to the necessary tasks and outputs. Using low-code components enables us to effectively direct validation tasks whilst allowing the LLM to comprehend the specifications and produce the required configurations and scripts, going beyond a simple template-based method. This strategy optimises pipeline creation and clearly defines all essential components and dependencies.

Our research focuses on the trustworthiness of software. LLMs often produce inaccurate output, which is detrimental to the development of trustworthy software. Using LLMs to generate computer code (such as MLOps pipelines) also carries the risk of runtime errors in the generated configuration file and embedded scripts, leading to unreliable systems.

Our novel approach addresses trustworthiness when using LLMs to automate the creation of MLOps pipelines by incorporating error detection and correction mechanisms to foster trust in our solution. Our approach is based on a low-code and template-based implementation using the Pipes and Filters pattern for modularity to steer and validate the LLM's results, meaning that the pipelines generated by the LLM are not only created and deployed, but also actively checked for correctness before they are used to train and deploy the RL model, contributing towards trustworthiness. This validation

and correction strategy directly increases trustworthiness by minimising errors in the underlying infrastructure and issues with LLM-based code generation. The low-code aspect makes the process accessible to people without in-depth DevOps knowledge.

10.4.4 Proposed Workflow

Figure 10.3 shows how a practitioner with a specific project can use our implemented approach. In the first step (**Write Initial Pipeline Specification**), the user writes an

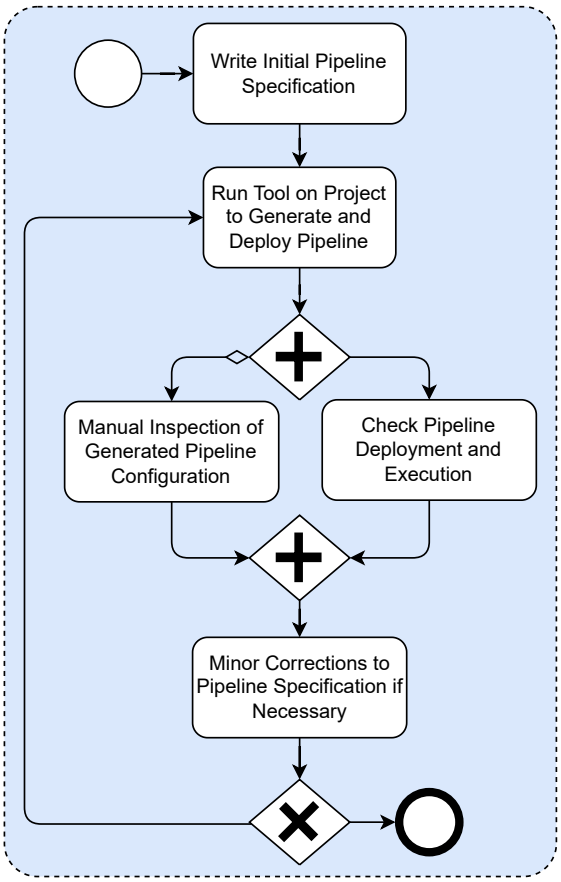

Figure 10.3: Workflow of the proposed approach.

initial version of the pipeline specification using our template described in Table 10.1. The practitioner runs our implementation on their project and specification file (**Run Tool on Project to Generate and Deploy Pipeline**). During this step, our program generates the GitLab CI/CD pipeline configuration file from the specification and automatically commits it to a GitLab repository. Optionally, the practitioner can then manually inspect the generated pipeline configuration (**Manual Inspection of Generated Pipeline Configuration**) whilst also verifying that the pipeline was deployed and executed, and that it functions correctly (**Check Pipeline Deployment and Execution**). If necessary,

the user can make corrections to the pipeline specification (**Minor Corrections to Pipeline Specification if Necessary**) and rerun our program to generate a new pipeline configuration.

The resulting workflow therefore incorporates an iterative feedback loop, allowing the user to incrementally, iteratively and continuously develop and refine the pipeline specification using our tool and refine the pipeline specification by repeating the process. The optional human-in-the-loop manual inspection and approval help build trust in the LLM-generated results and hold the human operator accountable. In reality, it is unlikely that many iterations would be needed, and in Sections 10.6 and 10.7 we provide results showing that if a suitable LLM is used, only very few minor corrections might be required.

10.4.5 Illustrative Example of the Approach in Action

To illustrate our approach, we present an example of generating and deploying an RL pipeline using a generation flow. An example of a concrete pipeline specification according to the template is found in Listing 10.1.

```
Usage:
python snake_rl.py ?train|eval|unit_test|model_test?

Required Files:
snake_env.py, snake_learn.py, snake_play.py,
snake_rl.py, snake_unit_test.py,
snake_test_model.py, snake_eval.py

Requirements:
gymnasium, numpy, opencv-python, stable-baselines3, tensorboard

Environment Variables:
ITERATIONS 2
RENDER_ASCII "True"
MODEL_NAME "LATEST"
MODELS_DIR "models/PPO-T4-1"
LOGS_DIR "logs/PPO-T4-1"
NUM_EPISODES 20

Gitlab CI pipeline stages:
unit_test, train, model_test, eval, deploy

Gitlab CI stage requirements:
- train creates artifacts at location MODELS_DIR
- eval job needs train job and its artifacts
- model_test job needs train job and its artifacts

Gitlab CI pipeline generates artifacts:
Yes, it creates artifacts for the models in the train stage

Gitlab CI pipeline generates docker images:
```

```
No

Details of the deploy stage:
-  Deployment to a local server "127.0.0.1".
-  The server is accessed by the user "user".
-  The private key is stored in the Gitlab variable SSH_PRIVATE_KEY.
-  Deployment tasks:
  -  Secure copy the files in MODELS_DIR to a directory "deploy_test"
  -  Run the model_test again on the server as a smoke test
```

Listing 10.1: An example low-code specification.

The requirements for generating the RL pipeline include a sample command to run the RL example's main script, a list of essential project files to be used across the various pipeline stages, the project's Python dependencies and the necessary environment variables with default values. Next, the GitLab CI/CD stages `unit_test`, `train`, `model_test`, `eval` and `deploy` are required in that order. The generation process should determine how to run the corresponding RL scripts. The `train` stage should provide models as artefacts for evaluation and model testing. Finally, deployment details specify the server and the private key (provided through a GitLab environment variable). Deployment commands for the `deploy` stage are also specified in natural language text.

Our tool generates a pipeline configuration YAML, assembles the project files in the generation directory and commits the entire project to GitLab. GitLab then runs the pipeline stages and, if successful, tests the code, trains and evaluates the models, deploys the trained models to the deployment server and runs a model test on the server as a smoke test.

10.4.6 Prompting Method

We use *few-shot prompting* [254] for each generation, detection and validation step. This method supports in-context learning by including examples in the prompt to guide the LLM towards better performance and by enabling it to produce results close to the desired outcome. When retries are needed, we provide the LLM with the previous conversation, including the generation history and any detected errors or warnings, which helps it refine its output to conform to the desired specification. In this manner, we implement *prompt chaining* [254] as a secondary prompt engineering technique. This structured approach ensures that each component of the generation flow is accurately produced and validated, resulting in a reliable, efficient pipeline.

Figure 10.4 depicts a conversation template that describes how our approach interacts with an LLM to generate a CI/CD pipeline configuration based on a user-authored low-code specification. Initially, our implementation specifies the **system** role when preparing the API call to the LLM, corresponding to **Role: System** in the **System Message Setup** box in Figure 10.4. Our implementation also prepares a system message instructing the LLM to create a GitLab CI/CD YAML file, which corresponds to the **Context Initialisation** in the **System Message Setup** box in Figure 10.4. Together with the

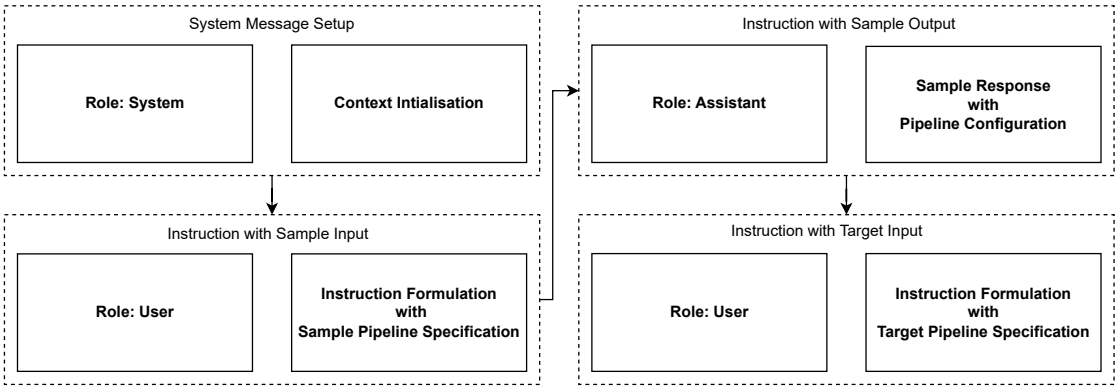

Figure 10.4: The approach's conversation template.

system role, the context initialisation is used to set the context for the instructions that are to follow, as shown in Listing 10.2.

```
You are a helpful code assistant that can teach a junior developer
    how to code. Your language of choice is Python and you want to
    create a Gitlab CI/CD yml file for your project.
Don't explain the code, just generate the code block itself.
```

Listing 10.2: Context initialisation provided to the LLM.

In the **Instruction with Sample Input** box in Figure 10.4, our implementation specifies the **user** role (**Role: User** in Figure 10.4). Setting the role to **user** indicates to the LLM that the message contains a request from the user that the LLM should respond to. Our tool also provides an instruction with an example pipeline specification (**Instruction Formulation with Sample Pipeline Specification** in Figure 10.4) as in Listing 10.3.

```
Write a Gitlab CI/CD yml file for Python machine-learning scripts.
Do not use Docker inside the Gitlab CI/CD pipeline. Use the following
    context:

Usage: python train_rl.py

Required Files:
train_rl.py
eval_rl.py
test_rl.py

Requirements:
stable-baselines3>=2.1.0
supersuit>=3.9.0

Gitlab CI/CD pipeline tasks:
train, eval, test

Gitlab CI/CD pipeline generates artifacts:
```

```
Yes, it creates artifacts for the models in the train stage

Gitlab CI/CD pipeline generates docker images:
No
```

Listing 10.3: An instruction and example low-code specification provided to the LLM.

We next set the role to **assistant** (**Role: Assistant** in the **Instruction with Sample Output** box in Figure 10.4), which communicates to the LLM that the text we are about to provide is to be considered a previous response from the LLM. Doing so makes the kind of response we expect to receive clear to the LLM. Specifically, we provide a sample output response for the sample input pipeline specification described previously (**Instruction with Sample Input** in Figure 10.4). This sample response contains a corresponding GitLab CI/CD pipeline configuration and is denoted by **Sample Response with Pipeline Configuration** in Figure 10.4. The sample response is as in Listing 10.4:

```
image: python:3.9-bullseye

stages:
  - train
  - eval
  - test

before_script:
  - python --version
  - pip install --upgrade pip
  - pip install stable-baselines3>=2.1.0 supersuit>=3.9.0

train:
  stage: train
  script:
    - python train_rl.py
  artifacts:
    paths:
      - path/to/model/directory/
    expire_in: 1 week

eval:
  stage: eval
  script:
    - python eval_rl.py

test:
  stage: test
  script:
    - python test_rl.py
```

Listing 10.4: An example CI/CD pipeline specification provided to the LLM.

Our implementation selects the **user** role again, indicating that we expect a response from the LLM (**Role: User** in **Instruction with Target Input** in Figure 10.4). We provide

the same instruction as in the **Instruction with Sample Input** part of Figure 10.4 (**`Write a Gitlab CI/CD yml file for Python machine-learning scripts. Do not use Docker inside the Gitlab CI/CD pipeline. Use the following context:`**), but instead of providing a sample pipeline specification, this time we give a target pipeline specification for which we wish to generate a real pipeline configuration (i.e. the output should be the contents of a GitLab CI/CD YAML configuration file). This step corresponds to **Instruction Formulation with Target Pipeline Specification** in **Instruction with Target Input** in Figure 10.4. The target specification takes the same format as the sample specification provided in **Instruction Formulation with Sample Pipeline Specification** in Figure 10.4. The specification format is described in Table 10.1 in Section 10.4.3.

The tool then constructs the conversation by combining the system message, the instruction with the example low-code, template-based pipeline specification, an example GitLab CI/CD YAML file response and the instruction with the target pipeline specification from which to generate the GitLab CI/CD YAML file configuration. This structured approach provides the LLM with unambiguous instructions and examples, increasing the likelihood of receiving an acceptable response.

Our program checks the LLM's response for GitLab CI/CD file linter issues, artefact creation and stage order correctness, and if errors or warnings occur, continues the conversation with the LLM by providing a **role: user** instruction to correct the detected errors and warnings in the generated configuration. It retries this generation process up to a predefined number of times (three by default) and finally selects the best response based on the number of errors and warnings encountered for each response. Doing so gives the LLM multiple opportunities to correct its output and allows our tool to select the best response from the LLM's proposed responses.

10.4.7 Generation Architecture

This section outlines the architecture of our approach to generating MLOps pipelines. We describe the system's primary components, classes and their roles below. Figure 10.5 shows the main high-level components of the architecture and Figure 10.6 shows the base classes containing the implementations for flows, code generation, validation and committing the generated results.

Components

Generator, Detector and LLM Components: `LCFExecutor` in Figure 10.5 is the central component responsible for executing the generation flow or running a detection flow. It uses instances of `LLM`, which wrap supported LLMs based on an `LLMDefinition`. An `LLMDefinition` specifies the model name, model type and parameters like token limit, minimum response tokens and temperature. The `LLM` component abstracts the functionalities of specific LLMs and facilitates the generation of textual artefacts based on provided specifications.

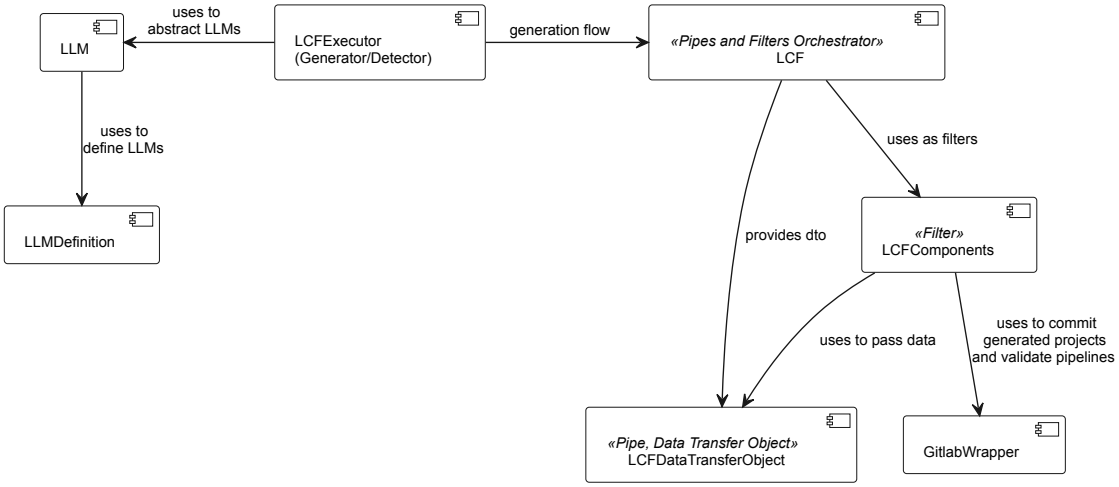

Figure 10.5: The main components of the generation architecture.

Pipes and Filters Architecture: The `LCF` component Pipes and Filters orchestrator in Figure 10.5 is responsible for executing `LCFComponents`, which represent different steps in the Pipes and Filters pattern involved in generation, detection and retrying. The `LCF` orchestrator also provides a `LCFDataTransferObject`, which is passed through the various steps in the Pipes and Filters pattern architecture. It carries the generation report, which is gradually assembled during the generation, detection and validation steps, metadata such as project and generation directories and the current conversation. This piping mechanism is based on passing a Data Transfer Object [285] through the implementation's filters. A `GitlabWrapper` component interacts with the GitLab server via its API to validate the generated CI/CD pipeline configuration files and upload them to GitLab.

Classes

Pipes and Filters Classes: Figure 10.6 shows the `LCFComponent` class, the base class for all classes involved in the generation and validation processes. Thus, its subclasses are the Pipes and Filters pattern architecture filters. `LCF` orchestrates a sequential flow of components, managing a list of `LCFComponent` components to define the sequence of operations, and an instance of `LCF` must be passed to either a generator or a detector for execution.

`LCFParallelFlow` subclasses `LCF`, managing the parallel execution of multiple flow steps and providing mechanisms for parallel validation via its subclass `ParallelDetection`, which is used for parallelising detection flows. `ParallelDetection` itself has two subclasses: `ParallelSuccessfulIfAllSuccessful`, which validates multiple tasks in parallel and is successful if all tasks succeed and `ParallelSuccessfulIfAnySuccessful`, which validates multiple tasks in parallel and is successful if any succeed.

`GenerationRetry` is also a subclass of `LCFComponent`. It retries generation steps a specified number of times, recording errors and warnings. It stops at the first attempt without errors or warnings; if no such attempt occurs, it takes the best results after the

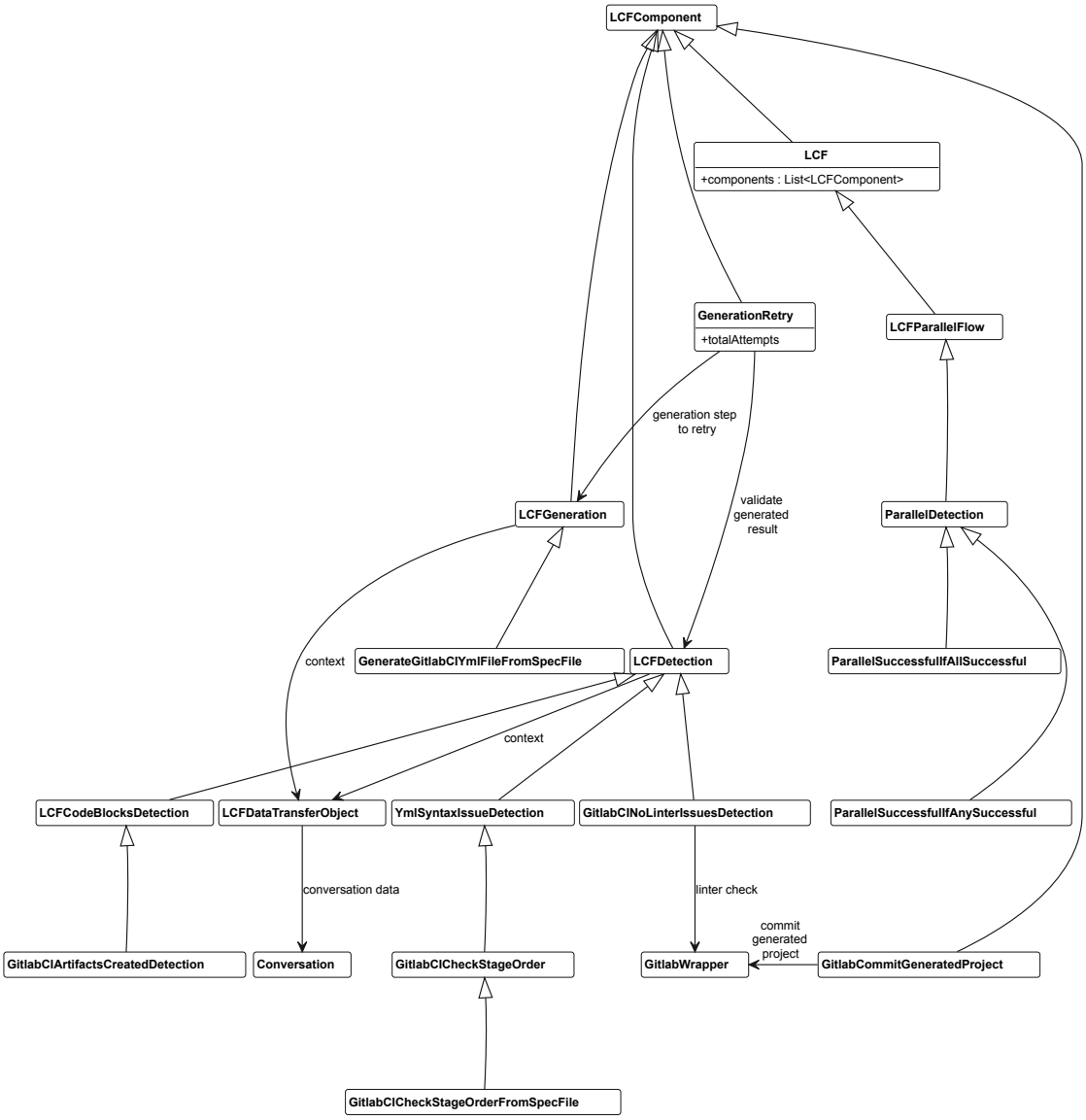

Figure 10.6: The base classes for flows, generation, detection and commits in `LCFComponent`.

specified total number of attempts (three, by default).

`GitlabCommitGeneratedProject` commits the generated pipeline and its project to GitLab using the `GitlabWrapper` class. `GitlabWrapper` provides an interface for interacting with a GitLab server's API to commit projects and validate GitLab pipelines.

Thus, the generation flow incorporates flexible components for parallel or concurrent validations (any or all of which must succeed, depending on the user's needs), retrying the generation and committing and pushing the project to GitLab.

Generation Classes: The `GenerationRetry` class shown in Figure 10.6 uses the `LCFGeneration` class, which is a base class used for performing generation tasks. `LCFGeneration` uses an `LCFDataTransferObject`, which in turn utilises the `Conversation` class, to manage interactions with the LLM by abstracting individual conversation instances, including messages to and responses from the LLMs and maintaining the context and state of ongoing interactions. The `GenerateGitlabCIYmlFileFromSpecFile` class subclasses `LCFGeneration` and is used to generate GitLab CI/CD YAML files from low-code specifications.

Detection Classes: Figure 10.6 shows the `LCFDetection` class, another subclass of `LCFComponent` used by `GenerationRetry`, which performs the detection used by validation tasks in generation workflows. It ensures that generated artefacts meet specified quality and correctness criteria for generation tasks. Like `LCFGeneration`, `LCFDetection` also makes use of a `LCFDataTransferObject`, which is passed through the Pipes and Filters flow with a `Conversation`.

Validation Classes: Figure 10.6 also includes various subclasses for performing individual validations. Several classes that subclass `LCFDetection` implement validation logic used during detection. For instance, `YmlSyntaxIssueDetection` is the base class for detecting issues in YAML files. Its subclass, `GitlabCICheckStageOrder`, ensures the stages in the GitLab CI pipeline are in the correct order, and the `GitlabCICheckStageOrderFromSpecFile` class validates the stage order against the low-code specification input. `GitlabCINoLinterIssuesDetection` uses a `GitlabWrapper` to validate the pipeline remotely via a GitLab server API call. `LCFCodeBlocksDetection` has a subclass, `GitlabCIArtifactsCreatedDetection`, that detects whether artefacts are created in the generated pipeline configuration.

10.4.8 Generation Flow Specification Approach

Using LLMs for code generation and validation presents accuracy challenges such as AI hallucinations and incorrect classifications [272], [273], [274]. Our approach employs a structured generation flow with focused steps and contextual feedback to address these issues and ensure reliable pipeline generation.

The generation flow specification for our approach is defined using a simple Python DSL. This specification guides the LLM through particular, focused generation, detection and validation steps to ensure high accuracy in the generated outcomes. An example of such a generation flow specification is shown in Listing 10.5:

```
LCF(
```

```
  components=[
    GenerationRetry(
      GenerateGitlabCIYmlFileFromSpecFile,
        ParallelSuccessfulIfAllSuccessful(
          [
            GitlabCINoLinterIssuesDetection,
            GitlabCIArtifactsCreatedDetection,
            GitlabCICheckStageOrderFromSpecFile
          ]
        )
      ),
      GitlabCommitGeneratedProject,
  ]
)
```

Listing 10.5: An example use of the Python DSL.

In the flow above, we specify that we wish to generate a GitLab CI/CD YAML file from a specification file (i.e. a low-code, template-based specification that describes the CI/CD stages and steps). We then wish to run various error detection measures in parallel: linting the generated GitLab pipeline configuration file, checking that the pipeline configuration creates artefacts, and verifying that the pipeline has the correct stage order. Finally, assuming all three detections passed, we wish to commit and push the project with its generated pipeline configuration to GitLab, where the pipeline is executed using that configuration.

10.5 Evaluation

We selected three open-source RL projects (see Section 10.5.1) and developed pipelines for each project, which were applied in three different scenarios (see Section 10.5.2) and multiple iterations. Each scenario has increasingly complex requirements to simulate real-world pipeline practices. This approach allows us to thoroughly evaluate our method's versatility, adaptability and scalability in addressing different pipeline requirements. By focusing on various scenarios for each project rather than simply increasing the number of projects, we capture a wider range of potential use cases and challenges, and ensure our approach is robust and effective in meeting both standard and unforeseen requirements in real-world applications.

10.5.1 Study Objects

Ideally, we would select a set of open-source RL projects that already include an MLOps pipeline. If these pipelines collectively contained all the pipeline stages our approach can generate, we could use them as the ground truth to assess the generation results. After an intensive search on GitHub and the Web, we found no open-source projects that fulfilled these requirements.

We then searched on GitHub, Google and Bing, with search terms including "(reinforcement learning | RL | multi-agent reinforcement learning | MARL | (Tensorflow & RL) | (Keras & RL) | RLLib | StableBaselines) & (MLOps | DevOps | CI/CD | Continuous Integration | Continuous Deployment | Continuous Delivery)". We found that there were not many repositories that combined these two fields.

As a second-best option for creating the ground truth, we selected three popular RL projects and implemented MLOps pipelines and various tests for them. We only used the first part of our search string (before the "&") and the following inclusion/exclusion criteria:

(1) Projects must primarily focus on RL.

(2) Projects have well-documented, clearly explained code that reflects the typical setup data scientists use for RL projects.

(3) Projects follow different approaches to constructing RL projects.

(4) All scenarios described in Section 10.5.2 are fully applicable.

We deliberately opted for several small projects rather than many large ones, as the specific RL implementation plays a secondary role in our evaluation. Instead, integrating these projects into each of the evaluation scenarios presented in Section 10.5.2 increases the complexity of the evaluation cases. Our approach focuses on MLOps components and treats the RL code primarily as a black box integrated into the pipeline stages – we are principally interested in MLOps at the level of pipeline architectures in this study, not the specifics of the implementations. Many larger, existing RL projects failed to meet inclusion/exclusion criterion number (4) ("All scenarios described in Section 10.5.2 are fully applicable"), meaning that only relatively trivial MLOps pipeline generation tasks would have been tested. Table 10.2 summarises the projects we included in our study.

The three projects were selected because they offer sufficient diversity in RL approaches whilst maintaining the rigour needed to perform a thorough evaluation using the evaluation scenarios. This focused approach allows us to conduct comprehensive, scenario-complete evaluations that would not be possible with a larger set of projects that only partially meet our evaluation criteria. The breadth vs depth trade-off aligns more with our research objectives: we are trying to validate MLOps generation capabilities, not survey all existing RL project structures. To demonstrate scalability and applicability beyond the three projects, we show how our MLOps pipeline generation can be integrated with larger, production-scale RL deployments in an industry application described in Section 10.8. This setting is a much more complex RL implementation than our evaluation projects, with multi-agent systems, distributed training architectures and enterprise-grade deployment requirements.

10.5.2 Scenarios

We define three evaluation scenarios, each subsequent scenario building on the previous one, for representing different MLOps pipeline practices, as well as the different kinds of generation requirements (precisely specified and natural language-based specifications):

Table 10.2: Descriptions of the Open-Source Projects

Name	***Description***
KAZ	A Stable-Baselines3 realisation of RL training, prediction, visual play and evaluation for the Petting Zoo Knights, Archers and Zombies environment. It uses the PPO algorithm, parallel environments and model saving/loading, and it can be customised using many parameters. Original implementation: `https://github.com/Farama-Foundation/PettingZoo/tree/master/pettingzoo/butterfly/knights_archers_zombies`
SNA	A Stable-Baselines3 realisation of RL training, prediction, visual play and monitoring for a custom snake game environment. It uses the PPO algorithm, model saving/loading, checkpoints, TensorBoard and a custom environment. Original implementation: `https://youtube.com/watch?v=XbWhJdQgi7E&list=PLQVvvaa0QuDf0O2DWwLZBfJeYY-JOeZB1`
SHO	A Keras/TensorFlow-based realisation of RL training, prediction and evaluation for a custom shower simulation environment. It uses the Deep Q-Network algorithm with a Boltzmann Q-policy, a sequential model and a custom environment. Original implementation: `https://github.com/nicknochnack/OpenAI-Reinforcement-Learning-with-Custom-Environment`

1. The existing basic functionality of the projects used as objects of our study is realised as an MLOps pipeline. This pipeline usually consists of training and evaluation stages, with several specific Python requirements that require setting environment variables and generating an RL model as an artefact during the training stage. Each generation requirement is precisely specified using a specification file such as the one in Section 10.4.5. Under the section `Gitlab CI stage requirements` in the specification, one natural language requirement is added that the training stage (`train`) creates artefacts in a models directory, and another one that the evaluation stage (`eval`) requires these artefacts (only if an evaluation stage is defined).

2. In addition to all generation requirements of the basic scenario, we require a deployment stage (`deploy`) and Docker images as container artefacts. We further detail this requirement under `Details of the deploy stage` in two natural language requirements: firstly, the deployment to a Docker container provided as an artefact by the `deploy` stage is required. Secondly, we require the `deploy` stage to use Docker in Docker (DIND) to create the Docker images. We specify, as further natural language requirements under `Gitlab CI stage requirements`, that the evaluation job requires the training job (and its artefacts). We also require an environment variable, *MODEL_DIR*, that specifies the model directory.

3. In this scenario, we use all generation requirements of the basic scenario and the natural language `Gitlab CI stage requirements` from the second scenario. We require five stages: `unit_test`, `train`, `model_test`, `eval` and `deploy`. We manually wrote unit and model tests for each project. These are placed in additional files that the pipeline must execute in the respective stages. Under `Gitlab CI stage requirements`, we add another natural language requirement: the `model_test` job

needs the `train` job and its artefacts. No Docker image creation is required. The deployment host and user names are provided under `Details of the deploy stage`. Further, it is stated that the private key to be used is stored in the GitLab variable *SSH_PRIVATE_KEY* and that secure copy is to be used as a deployment task to copy the artefact in *MODEL_DIR* to a directory on the host.

We executed each scenario three times per project, for a total of 27 evaluations per LLM. Our choice of evaluation scenarios was driven by the need to comprehensively evaluate MLOps pipeline generation over the most critical and commonly needed pipeline stages in RL deployments. The three scenarios described in this section were carefully designed, based on our understanding of RL and informed by discussions with our industry partner, Siemens, to capture fundamental MLOps practices when building CI/CD pipelines. These scenarios and their assorted evaluation variables include the key pipeline components – model training orchestration, hyperparameter optimisation workflows and deployment automation – where MLOps tools provide the most value.

We considered expanding and customising the scenarios to cover a broader scope of RL projects, but decided that doing so would dilute the evaluation's validity and depth. Such evaluation scenarios would not have tested our solution's ability to produce advanced pipeline stages that reflect the real complexity of production RL deployments. We focused on depth and real-world applicability over sample size, so our evaluation reflects real MLOps challenges rather than examples customised to existing projects.

10.5.3 Evaluation Variables

We use twenty evaluation variables to analyse the generated MLOps pipelines in our evaluation process; all variables have Boolean values corresponding to 1 or 0 representing success or failure, respectively. Some variables have the value "NA" if not required by the evaluation scenario's requirements.

We evaluated the following generation-specific variables by inspecting the generated pipeline configuration files:

- *GPI*: The generated script uses a correct Python image.
- *GAS*: The generated script contains all required stages.
- *GSO*: The required stages are in the correct order.
- *GAR*: All Python requirements and the necessary Linux packages are installed for all required stages correctly.
- *GEN*: All specified environment variables are set correctly.
- *GMS*: All model artefacts are created correctly.
- *GCD*: Required Docker images are created correctly.
- *GDA*: All Docker artefacts are created correctly.

We evaluated the following variables by inspecting the code to check whether the natural language text requirements under `Gitlab CI stage requirements` and `Details of the deploy stage` are realised correctly. Note that many other details could be provided in natural language; in our evaluation, to systematise our study, we limit ourselves to the following cases:

- *NTA*: The `train` job creates artefacts in a specific directory.
- *NET*: The `eval` job needs the `train` job and its artefacts.
- *NMT*: The `model_test` job needs the `train` job and its artefacts.
- *NSK*: An SSH key setup script is created correctly for reaching the deployment server.
- *NSC*: Secure copy is used to perform the deployment with the correct user/key, host and directory.
- *NDA*: The `deploy` stage provides Docker results as an artefact.
- *NDI*: Docker in Docker (DIND) is realised, and the correct image is selected.

Finally, we evaluated the following runtime variables to determine whether a pipeline job ran successfully.

- *RUT*: Unit tests ran successfully.
- *RTR*: Model training ran successfully.
- *RTE*: Model tests ran successfully.
- *REV*: Model evaluation ran successfully.
- *RDP*: Model deployment ran successfully.

We executed each scenario three times for each study object and each LLM. We then calculated the error rate for each evaluation variable *var* as given by Equation 10.1:

$$\overline{E}_{\text{var}} = 1 - \frac{1}{n}\sum_{i=1}^{n} S^{i}_{\text{var}} \tag{10.1}$$

where n is the number of iterations ($n = 3$ in this case study) and S^{i}_{var} is the success rate (valued 0 or 1) for the variable *var* at iteration i.

We also calculated the average error rate over all precisely specified generation variables (*Average Error GS*), all variables relating to natural language-based requirements (*Average Error NL*), all variables associated with running the jobs (*Average Error RJ*) and all variables (*Average Error*) using Equation 10.2:

$$\overline{E}_{\text{cat}} = 1 - \frac{1}{nm}\sum_{i=1}^{n}\sum_{j=1}^{m} S^{i,j} \tag{10.2}$$

where n is the number of iterations ($n = 3$), m is the number of variables in the category ($m = 8$ for precisely-defined generation tasks, $m = 7$ for natural language tasks, and $m = 5$ for runtime tasks) and $S^{i,j}$ is the success rate (valued 0 or 1) on iteration i for variable j in the category.

10.5.4 Models

We chose seven popular and current LLMs for evaluating our approach: **OpenAI GPT-3.5 Turbo**, **OpenAI GPT-4o**, **Qwen2.5 72B**, **Qwen2.5 Coder 32B**, **Llama 3.3 70B**, **Code Llama 70B** and **DeepSeek R1 70B** [286]. We deployed and ran the two OpenAI models on Microsoft Azure. We deployed and ran the five freely available models in Ollama on a Cisco UCS C245 M6 server with an NVIDIA A100 80GB GPU, two AMD EPYC 7543 32-core processors and 512GB of RAM.

The five Ollama models are of particular interest because they represent a growing cohort of small, performant LLMs that demonstrate reasonable capabilities for solving tasks but are freely available and locally deployable thanks to their relatively low memory footprint and flexible parameter sizes, which can be selected based on local hardware capabilities. These locally executable LLM types expand the range of applied model types beyond that of the GPT family.

The LLMs were selected from the standard benchmark EvalPlus [287][11], which is based on peer-reviewed studies and provides a ranked list of code generation LLMs, including versions of the ones we use in this study. Where a newer or higher-parameter version of a model was available, we selected that version within the constraints of our hardware. We only selected popular models, i.e. models with over one million pulls on Ollama. Furthermore, similarly to how we chose GPT-3.5 Turbo and then an enhanced version in GPT-4o, we selected base models (Qwen2.5 72B and Llama 3.3 70B) and corresponding coding-specific related models (Qwen2.5 Coder 32B and CodeLlama 70B). We also included DeepSeek R1 70B due to its novelty and popularity.

The LLMs are widely used for related but distinct tasks and exhibit varying characteristics, including speed, contextual awareness and training data. Even though it may appear obvious that some LLMs are likely to perform much better than others, this can only be confirmed by evaluating a specific application. Evaluating a selection of LLMs enables us to compare how they affect the results of our approach.

GPT-3.5 Turbo is a widely adopted, efficient model from OpenAI. GPT-3.5 Turbo delivers an effective baseline for assessing our approach because it strikes the right equilibrium between operational efficiency and performance quality. GPT-3.5 Turbo is a standard reference model to check progress in new technologies or customised coding solutions. The model is more advanced than previous versions, although it sometimes exhibits diminished mathematical capabilities. We included GPT-3.5 Turbo as a model

[11] Also see `https://trustbit.tech/en/llm-leaderboard-juli-2024`, `https://klu.ai/llm-leaderboard`, `https://aimlapi.com/comparisons/chatgpt-4o-vs-o1-mini`, `https://the-decoder.com/gpt-4-crushes-other-llms-according-to-new-benchmark-suite` and links on EvalPlus for further examples of benchmarks.

for assessment, since recent benchmarks demonstrate its comparable performance to the top non-commercial LLM models, but at substantially reduced cost compared to GPT-4o.

GPT-4o from OpenAI delivers enhanced speed and efficiency, and is one of the best models in the EvalPlus performance benchmark. The model exhibits powerful general capabilities, making it ideal for state-of-the-art code generation. The capabilities of GPT-4o extend to basic coding and general awareness tasks, but this advanced performance level comes at a greater cost than other models. Our evaluation also uses GPT-4o because of its superior ability to understand and generate natural language, which underpins our methodology.

Qwen2.5 72B, created by Alibaba Cloud, has over 4.1 million pulls as of 2025-02-14 and a size of 47GB. We use Qwen2.5 in the evaluation to assess the effectiveness of our approach, using a model that demonstrates superior multilingual skills, enhanced coding capabilities, advanced instruction-following and support for structured text data understanding and generation.

Qwen2.5 Coder 32B by Alibaba Cloud has over 2.2 million pulls as of 2025-02-14 and a size of 20GB. This model has been ranked highly across various popular code generation benchmarks, and its performance is comparable to GPT-4o. Qwen2.5 Coder 32B can also fix code errors, which is relevant to our approach's error-handling aspect. The model supports over 40 programming languages.

Llama 3.3 70B is a leading open-source model from Meta and is widely used, with over 1.3 million pulls as of 2025-02-14 and is 43GB in size. Llama 3.3 70B performs similarly to Llama 3.1 405B but with fewer parameters. It demonstrates particular strength when tasks require extended context, like document analysis and conversations spanning multiple exchanges, making it a suitable model for evaluating our approach.

Code Llama 70B from Meta, with over 1.7 million pulls as of 2025-02-14 and a size of 39GB, is well-suited to this study since it has been specifically trained for code generation tasks. The model supports most popular programming languages, including Python, C++, Java, PHP, TypeScript, JavaScript, C# and Bash.

DeepSeek R1 70B [286] is a new LLM released in January 2025 with over 15.6 million pulls as of 2025-02-14 and a size of 43GB. We chose this LLM due to its novelty and the high benchmark ranking of its predecessor (DeepSeek V3 from November 2024), and opted for a parameter size of 70B. It is a distilled model based on Llama3.3 70B. Early benchmarks suggest it should be a high-performing model, and evaluation results by DeepSeek and EvalPlus suggest that its base model (DeepSeek V3) performs comparably to OpenAI O1 and OpenAI O1 Mini, which came first and second place on the EvalPlus leaderboard, respectively.

10.6 Initial Results

In this section, we interpret the evaluation results shown in Tables 10.3, 10.4, 10.5, 10.6, 10.7, 10.8 and 10.9. Please note that, based on these results, we also implemented some improvements to the project and ran a new evaluation, which we report on in Section 10.7. A discussion of the implications in the context of the research questions

can be found in Section 10.9.

The tables contain the evaluation results for each model for the scenarios and evaluation variables. The rows of each table specify an evaluation variable or task type as described in Section 10.5.3, such as *GPI* ("The generated script uses a correct Python image"). The next rows correspond to the sets of evaluation variable types, for example, *Average Error GS* represents all generation tasks. The final row, *Average Error*, represents all evaluation variables. The columns, except the final column, contain the error rates for the open-source projects *KAZ*, *SNA* and *SHO* described in Table 10.2 and the evaluation scenarios 1, 2 and 3 described in Section 10.5.2, as numerical suffixes. For instance, *SNA1*, *SNA2* and *SNA3* indicate the *SNA* project and each of the three evaluation scenarios, respectively. The last column, *Average Error*, shows the average error rates across all projects and scenarios for a given evaluation variable, evaluation category, or all evaluation variables. As another example, in Table 10.3, the cell represented by row *GEN* and column *KAZ 2* shows an error rate of 0.33 for the evaluation variable *GEN* ("All specified environment variables are set correctly") for project *KAZ* and scenario 2. The final column of the same row indicates an average error rate of 0.22 across all projects and scenarios for evaluation variable *GEN*. The cell for column *SHO 1* and row *Average Error GS* indicates an average error of 0.06 for that project and scenario for generation tasks, and the final row in that column shows an average error of 0.17 for project and scenario *SHO 1* across all evaluation variables.

10.6.1 Best and Worst-Performing LLMs

Table 10.10 summarises the best-performing LLMs by task type error rate, project and evaluation scenario, and average error rates.

The rows signify three different sets of error rates depending on the task type: *Precisely-Defined Generation Tasks*, which represent the average error rate for generation tasks, *Natural Language Tasks*, which denote the average error rate for natural language tasks and *Runtime Tasks*, which encapsulate the average error rate for runtime job tasks. The final row, *Average Error Rate*, aggregates the aforementioned error rates across task types (generation, natural language and runtime) for each column.

Except for the final one, the columns represent the error rates for the open-source projects *KAZ*, *SNA* and *SHO* described in Table 10.2 and combined with the evaluation scenarios 1, 2 and 3 described in Section 10.5.2, represented as numerical suffixes. For example, *KAZ1*, *KAZ2* and *KAZ3* refer to the *KAZ* project and each of the three evaluation scenarios. The final column, *Average Error Rate*, gives the average error rate across all projects and scenarios for a given task category. For example, the row and column pair corresponding to row *Natural Language Tasks* and column *KAZ3* indicates that GPT-4o had the lowest average error rate for natural language tasks for project *KAZ* and scenario 3. Row *Average Error Rate* and column *SHO 1* show that Qwen2.5 Coder and Llama 3.3 achieved the lowest average error rates across all task types for project *SHO* and scenario 1. The cell matching row *Runtime Tasks* and column *Average Error Rate* indicates that GPT-4o had the lowest average error rate for runtime tasks across all projects and scenarios.

Table 10.3: GPT-3.5 Turbo 0613 (Version 2023-06-13) Initial Results: Error Rates

	KAZ1	***KAZ2***	***KAZ3***	***SNA1***	***SNA2***	***SNA3***	***SHO1***	***SHO2***	***SHO3***	***Average Error***
GPI	0.00	0.67	0.00	0.00	0.33	0.00	0.00	0.00	0.00	0.11
GAS	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
GSO	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
GAR	0.00	0.00	0.00	0.00	0.33	0.00	0.00	0.67	0.00	0.11
GEN	0.00	0.33	0.33	0.33	0.00	0.33	0.33	0.00	0.33	0.22
GMS	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
GCD	NA	0.00	NA	NA	0.00	NA	NA	0.00	NA	0.00
GDA	NA	0.33	NA	NA	0.67	NA	NA	0.33	NA	0.44
NTA	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
NET	0.00	0.00	0.00	NA	0.00	0.00	1.00	0.00	0.00	0.11
NMT	NA	NA	1.00	NA	NA	1.00	NA	NA	0.67	0.89
NSK	NA	NA	1.00	NA	NA	0.33	NA	NA	0.33	0.56
NSC	NA	NA	1.00	NA	NA	1.00	NA	NA	1.00	1.00
NDA	NA	0.33	NA	NA	0.67	NA	NA	0.33	NA	0.44
NDI	NA	0.67	NA	NA	0.33	NA	NA	0.33	NA	0.44
RUT	NA	NA	0.00	NA	NA	0.00	NA	NA	0.67	0.22
RTR	0.00	0.67	0.00	0.00	0.33	0.00	0.00	0.67	0.00	0.19
RTE	NA	NA	0.00	NA	NA	0.00	NA	NA	0.67	0.22
REV	0.00	0.67	0.00	NA	0.33	0.33	0.33	0.00	0.00	0.19
RDP	NA	1.00	1.00	NA	0.67	1.00	NA	0.33	1.00	0.83
Average Error GS	0.00	0.17	0.06	0.06	0.17	0.06	0.06	0.13	0.06	0.08
Average Error NL	0.00	0.25	0.60	0.00	0.25	0.47	0.50	0.17	0.40	0.29
Average Error RJ	0.00	0.78	0.20	0.00	0.44	0.27	0.17	0.33	0.47	0.30
Average Error	0.00	0.31	0.27	0.04	0.24	0.25	0.17	0.18	0.29	0.19

Table 10.4: GPT-4o (Version 2024-08-06) Initial Results: Error Rates

	KAZ1	***KAZ2***	***KAZ3***	***SNA1***	***SNA2***	***SNA3***	***SHO1***	***SHO2***	***SHO3***	***Average Error***
GPI	0.00	0.33	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.04
GAS	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
GSO	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
GAR	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
GEN	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
GMS	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
GCD	NA	0.00	NA	NA	0.00	NA	NA	0.00	NA	0.00
GDA	NA	0.00	NA	NA	0.00	NA	NA	0.00	NA	0.00
NTA	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
NET	0.00	0.00	0.00	NA	0.00	0.00	1.00	0.00	0.00	0.11
NMT	NA	NA	0.00	NA	NA	0.00	NA	NA	0.00	0.00
NSK	NA	NA	0.00	NA	NA	0.33	NA	NA	0.00	0.11
NSC	NA	NA	0.00	NA	NA	0.33	NA	NA	0.00	0.11
NDA	NA	0.00	NA	NA	0.00	NA	NA	0.00	NA	0.00
NDI	NA	0.00	NA	NA	0.33	NA	NA	0.33	NA	0.22
RUT	NA	NA	0.33	NA	NA	0.00	NA	NA	0.33	0.22
RTR	0.00	0.33	0.33	0.00	0.00	0.00	0.00	0.00	0.33	0.11
RTE	NA	NA	0.33	NA	NA	1.00	NA	NA	0.33	0.56
REV	0.00	0.33	0.33	NA	0.00	0.33	0.00	0.00	0.33	0.15
RDP	NA	1.00	0.33	NA	1.00	1.00	NA	1.00	0.33	0.78
Average Error GS	0.00	0.04	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Average Error NL	0.00	0.00	0.00	0.00	0.08	0.13	0.50	0.08	0.00	0.09
Average Error RJ	0.00	0.56	0.33	0.00	0.33	0.47	0.00	0.33	0.33	0.26
Average Error	0.00	0.13	0.10	0.00	0.09	0.19	0.10	0.09	0.10	0.09

Table 10.5: Qwen2.5 72B Initial Results: Error Rates

	KAZ1	***KAZ2***	***KAZ3***	***SNA1***	***SNA2***	***SNA3***	***SHO1***	***SHO2***	***SHO3***	***Average Error***
GPI	0.00	0.00	0.00	0.00	0.33	0.00	0.00	0.00	0.00	0.04
GAS	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
GSO	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
GAR	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
GEN	0.00	0.33	0.67	0.00	0.00	0.33	0.00	0.00	1.00	0.26
GMS	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
GCD	NA	0.00	NA	NA	0.33	NA	NA	0.00	NA	0.11
GDA	NA	0.00	NA	NA	0.00	NA	NA	0.00	NA	0.00
NTA	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
NET	0.00	0.00	0.00	NA	0.00	0.00	1.00	0.00	0.00	0.11
NMT	NA	NA	0.00	NA	NA	0.00	NA	NA	0.00	0.00
NSK	NA	NA	0.00	NA	NA	0.33	NA	NA	0.00	0.11
NSC	NA	NA	0.33	NA	NA	0.67	NA	NA	0.33	0.44
NDA	NA	0.00	NA	NA	0.00	NA	NA	0.00	NA	0.00
NDI	NA	0.67	NA	NA	1.00	NA	NA	0.67	NA	0.78
RUT	NA	NA	1.00	NA	NA	0.33	NA	NA	1.00	0.78
RTR	0.00	0.00	1.00	0.00	0.33	0.33	0.00	0.00	1.00	0.30
RTE	NA	NA	1.00	NA	NA	0.33	NA	NA	1.00	0.78
REV	0.00	0.33	1.00	NA	0.33	0.67	0.00	0.00	1.00	0.37
RDP	NA	0.67	1.00	NA	1.00	1.00	NA	1.00	1.00	0.94
Average Error GS	0.00	0.04	0.11	0.00	0.08	0.06	0.00	0.00	0.17	0.05
Average Error NL	0.00	0.17	0.07	0.00	0.25	0.20	0.50	0.17	0.07	0.16
Average Error RJ	0.00	0.33	1.00	0.00	0.56	0.53	0.00	0.33	1.00	0.42
Average Error	0.00	0.13	0.38	0.00	0.22	0.25	0.10	0.11	0.40	0.18

Table 10.6: Qwen2.5 Coder 32B Initial Results: Error Rates

	KAZ1	***KAZ2***	***KAZ3***	***SNA1***	***SNA2***	***SNA3***	***SHO1***	***SHO2***	***SHO3***	***Average Error***
GPI	0.00	0.33	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.04
GAS	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
GSO	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
GAR	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
GEN	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.11
GMS	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
GCD	NA	0.00	NA	NA	0.67	NA	NA	0.00	NA	0.22
GDA	NA	0.33	NA	NA	0.00	NA	NA	0.33	NA	0.22
NTA	0.33	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.04
NET	0.00	0.00	0.00	NA	0.00	0.00	0.67	0.00	0.00	0.07
NMT	NA	NA	0.00	NA	NA	0.00	NA	NA	0.00	0.00
NSK	NA	NA	0.00	NA	NA	0.33	NA	NA	0.67	0.33
NSC	NA	NA	0.33	NA	NA	0.00	NA	NA	0.00	0.11
NDA	NA	0.67	NA	NA	0.67	NA	NA	0.33	NA	0.56
NDI	NA	0.33	NA	NA	0.67	NA	NA	0.33	NA	0.44
RUT	NA	NA	0.00	NA	NA	0.00	NA	NA	1.00	0.33
RTR	0.00	0.33	0.00	0.00	1.00	0.00	0.00	0.00	1.00	0.26
RTE	NA	NA	0.00	NA	NA	1.00	NA	NA	1.00	0.67
REV	0.00	0.33	0.00	NA	1.00	1.00	0.00	0.00	1.00	0.37
RDP	NA	0.67	0.33	NA	1.00	1.00	NA	0.33	1.00	0.72
Average Error GS	0.00	0.08	0.00	0.00	0.08	0.00	0.00	0.04	0.17	0.04
Average Error NL	0.17	0.25	0.07	0.00	0.33	0.07	0.33	0.17	0.13	0.17
Average Error RJ	0.00	0.44	0.07	0.00	1.00	0.60	0.00	0.11	1.00	0.36
Average Error	0.03	0.20	0.04	0.00	0.33	0.21	0.07	0.09	0.42	0.15

Table 10.7: Llama 3.3 70B Initial Results: Error Rates

	KAZ1	***KAZ2***	***KAZ3***	***SNA1***	***SNA2***	***SNA3***	***SHO1***	***SHO2***	***SHO3***	***Average Error***
GPI	0.00	1.00	0.00	0.00	1.00	1.00	0.00	0.00	0.00	0.33
GAS	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
GSO	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
GAR	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
GEN	0.00	0.00	0.00	0.33	0.33	0.33	0.00	0.00	0.00	0.11
GMS	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
GCD	NA	0.00	NA	NA	0.00	NA	NA	0.00	NA	0.00
GDA	NA	0.00	NA	NA	0.00	NA	NA	0.00	NA	0.00
NTA	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
NET	0.00	0.00	0.00	NA	0.00	0.00	0.67	0.00	0.00	0.07
NMT	NA	NA	0.00	NA	NA	0.00	NA	NA	0.00	0.00
NSK	NA	NA	0.33	NA	NA	0.00	NA	NA	0.00	0.11
NSC	NA	NA	0.00	NA	NA	0.00	NA	NA	0.00	0.00
NDA	NA	0.00	NA	NA	0.67	NA	NA	0.00	NA	0.22
NDI	NA	1.00	NA	NA	1.00	NA	NA	1.00	NA	1.00
RUT	NA	NA	0.00	NA	NA	1.00	NA	NA	0.67	0.56
RTR	0.00	1.00	0.00	0.00	1.e00	1.00	0.00	0.00	0.67	0.41
RTE	NA	NA	0.00	NA	NA	1.00	NA	NA	0.67	0.56
REV	0.00	1.00	0.00	NA	1.00	1.00	0.00	0.00	0.67	0.41
RDP	NA	1.00	0.00	NA	1.00	1.00	NA	1.00	0.67	0.78
Average Error GS	0.00	0.13	0.00	0.06	0.17	0.22	0.00	0.00	0.00	0.06
Average Error NL	0.00	0.25	0.07	0.00	0.42	0.00	0.33	0.25	0.00	0.15
Average Error RJ	0.00	1.00	0.00	0.00	1.00	1.00	0.00	0.33	0.67	0.44
Average Error	0.00	0.33	0.02	0.04	0.40	0.40	0.07	0.13	0.21	0.18

Table 10.8: Code Llama 70B Initial Results: Error Rates

	KAZ1	***KAZ2***	***KAZ3***	***SNA1***	***SNA2***	***SNA3***	***SHO1***	***SHO2***	***SHO3***	***Average Error***
GPI	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
GAS	0.00	0.00	0.00	0.00	0.00	0.00	0.67	0.00	0.00	0.07
GSO	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.33	0.04
GAR	0.00	0.00	0.00	0.00	0.00	0.67	0.00	0.33	0.00	0.11
GEN	1.00	0.67	0.67	0.00	0.33	0.67	0.00	0.67	0.67	0.52
GMS	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
GCD	NA	0.67	NA	NA	0.33	NA	NA	0.00	NA	0.33
GDA	NA	1.00	NA	NA	1.00	NA	NA	1.00	NA	1.00
NTA	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
NET	0.00	0.00	0.00	NA	0.33	0.33	1.00	0.00	0.33	0.22
NMT	NA	NA	0.00	NA	NA	0.33	NA	NA	0.00	0.11
NSK	NA	NA	0.33	NA	NA	0.33	NA	NA	0.33	0.33
NSC	NA	NA	0.00	NA	NA	1.00	NA	NA	0.33	0.44
NDA	NA	1.00	NA	NA	1.00	NA	NA	1.00	NA	1.00
NDI	NA	0.33	NA	NA	0.67	NA	NA	0.33	NA	0.44
RUT	NA	NA	0.67	NA	NA	0.67	NA	NA	0.33	0.56
RTR	0.00	0.67	0.67	0.00	0.67	0.67	0.00	0.33	0.33	0.37
RTE	NA	NA	0.67	NA	NA	0.67	NA	NA	0.33	0.56
REV	1.00	1.00	0.67	NA	1.00	1.00	1.00	1.00	0.33	0.78
RDP	NA	1.00	0.67	NA	1.00	1.00	NA	1.00	1.00	0.94
Average Error GS	0.17	0.29	0.11	0.00	0.21	0.22	0.11	0.25	0.17	0.17
Average Error NL	0.00	0.33	0.07	0.00	0.50	0.40	0.50	0.33	0.20	0.26
Average Error RJ	0.50	0.89	0.67	0.00	0.89	0.80	0.50	0.78	0.47	0.61
Average Error	0.20	0.42	0.27	0.00	0.42	0.46	0.27	0.38	0.27	0.30

Table 10.9: DeepSeek R1 70B (Version 2025-01-20) Initial Results: Error Rates

	KAZ1	***KAZ2***	***KAZ3***	***SNA1***	***SNA2***	***SNA3***	***SHO1***	***SHO2***	***SHO3***	***Average Error***
GPI	0.00	0.33	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.04
GAS	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
GSO	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
GAR	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
GEN	0.67	0.33	0.33	0.00	0.33	0.00	0.67	0.00	0.00	0.26
GMS	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
GCD	NA	0.00	NA	NA	0.33	NA	NA	0.00	NA	0.11
GDA	NA	0.33	note that,NA	NA	0.33	NA	NA	0.67	NA	0.44
NTA	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
NET	0.00	0.00	0.00	NA	0.00	0.00	0.33	0.00	0.00	0.04
NMT	NA	NA	0.00	NA	NA	0.00	NA	NA	0.00	0.00
NSK	NA	NA	0.67	NA	NA	0.33	NA	NA	0.67	0.56
NSC	NA	NA	0.33	NA	NA	0.33	NA	NA	0.67	0.44
NDA	NA	0.67	NA	NA	1.00	NA	NA	0.67	NA	0.78
NDI	NA	1.00	NA	NA	1.00	NA	NA	0.33	NA	0.78
RUT	NA	NA	0.33	NA	NA	0.33	NA	NA	1.00	0.56
RTR	0.00	0.33	0.33	0.00	0.67	0.67	0.00	0.00	1.00	0.33
RTE	NA	NA	0.33	NA	NA	0.67	NA	NA	1.00	0.67
REV	0.67	0.67	0.33	NA	0.67	0.67	0.67	0.00	1.00	0.52
RDP	NA	1.00	1.00	NA	1.00	1.00	NA	1.00	1.00	1.00
Average Error GS	0.11	0.13	0.06	0.00	0.13	0.00	0.11	0.08	0.00	0.07
Average Error NL	0.00	0.42	0.20	0.00	0.50	0.13	0.17	0.25	0.27	0.21
Average Error RJ	0.33	0.67	0.47	0.00	0.78	0.67	0.33	0.33	1.00	0.51
Average Error	0.13	0.31	0.23	0.00	0.36	0.25	0.17	0.18	0.40	0.22

Table 10.10: Summary of the Best-Performing LLMs by Lowest or Equal-Lowest Error Rate: Initial Results

	KAZ1	***KAZ2***	***KAZ3***	***SNA1***	***SNA2***	***SNA3***	***SHO1***	***SHO2***	***SHO3***	***Average Error Rate***
Precisely-Defined Generation Tasks	GPT-3.5 GPT-4o Qwen2.5 Qwen2.5 Coder Llama 3.3	GPT-4o Qwen2.5	GPT-4o Qwen2.5 Coder Llama 3.3	GPT-4o Qwen2.5 Qwen2.5 Coder Code Llama DeepSeek	GPT-4o	GPT-4o Qwen2.5 Qwen2.5 Coder DeepSeek	GPT-4o Qwen2.5 Qwen2.5 Coder Llama 3.3	GPT-4o Qwen2.5 Llama 3.3	GPT-4o Llama 3.3 DeepSeek	GPT-4o
Natural Language Tasks	GPT-3.5 GPT-4o Qwen2.5 Llama 3.3 Code Llama DeepSeek	GPT-4o	GPT-4o	All Equal	GPT-4o	Llama 3.3	DeepSeek	GPT-4o	GPT-4o Llama 3.3	GPT-4o
Runtime Tasks	GPT-3.5 GPT-4o Qwen2.5 Qwen2.5 Coder Llama 3.3	Qwen2.5	Llama 3.3	All Equal	GPT-4o	GPT-3.5	GPT-4o Qwen2.5 Qwen2.5 Coder Llama 3.3	Qwen2.5 Coder	GPT-4o	GPT-4o
Average Error Rate	GPT-3.5 GPT-4o Qwen2.5 Llama 3.3	GPT-4o Qwen2.5	Llama 3.3	GPT-4o Qwen2.5 Qwen2.5 Coder Code Llama DeepSeek	GPT-4o	GPT-4o	Qwen2.5 Coder Llama 3.3	GPT-4o Qwen2.5 Coder	GPT-4o	GPT-4o

Our initial approach achieved an average error rate of 0.187 across all seven LLMs. Overall, GPT-4o exhibited the lowest error rate, followed by Qwen2.5 Coder. GPT-4o also performed best for generation tasks, natural language tasks and runtime tasks, followed by Qwen2.5 Coder, Llama 3.3 and GPT-3.5 Turbo, respectively. Therefore, GPT-4o was the best overall performer across all categories, with Qwen2.5 Coder coming second overall and performing particularly well on code generation tasks.

The worst-performing LLM overall was Code Llama, followed by DeepSeek. Code Llama performed worst for code generation tasks, followed by GPT-3.5 Turbo. Natural language tasks were performed worst by GPT-3.5 Turbo, followed by Code Llama. Code Llama performed worst in runtime tasks, followed by DeepSeek. Hence, Code Llama performed poorly at code generation and natural language tasks.

In conclusion, our approach used GPT-4o and Qwen2.5 Coder to successfully generate GitLab CI/CD pipeline configuration files with low error rates. Note that the error rates reported in the tables in this section were easily improved further by resolving the common errors described in Section 10.6.4 – see Section 10.7 for details.

It should also be noted that the results we report are likely an underestimation of the LLMs' capabilities and potential in this application and an overestimation of the error rates, since if a pipeline stage failed, we also marked all subsequent pipeline stages as having failed. In reality, some of these pipeline stages would have succeeded, assuming the error in the previous stage had not occurred. Also, it was often the case that multiple errors were related and had a single cause. For instance, secure copy during deployment might have failed because the SSH key was not set up correctly, or a runtime error could have occurred due to an incorrect value in the pipeline configuration caused by a generation error. Both scenarios would increase the error rate across multiple evaluation variables but require only a single fix.

10.6.2 Evaluation Variables

Across all seven LLMs, some common patterns emerged in the results, particularly considering the subtasks the LLMs could reliably perform and where they often made mistakes. Generally, as might be expected, simpler tasks yielded lower error rates, whilst more complex ones yielded higher error rates. The tasks that most LLMs performed most accurately were generative tasks and a large proportion of natural language processing tasks. On the other hand, runtime tasks were more challenging for all of the LLMs. In Section 10.6.4, we discuss common errors in output that occurred, which were easily resolved by providing slightly more (and more specific) information to the LLM.

Generative tasks that the models mostly completed with low error rates included, for instance, using the correct Python image in the generation pipeline configuration file, including all the required stages in the proper order and installing necessary Python and Linux requirements. Some deviation in behaviour occurred when specifying environment variables and the correct Docker artefacts.

Several LLMs had difficulties with natural language tasks such as setting up SSH keys for model deployment, using secure copy to perform the deployment, understanding the use of *Docker in Docker* (DIND), selecting the correct Docker image and providing the

generated Docker image as a result. Problematic runtime tasks for several models were the successful execution of unit and model tests, and deploying the model. Failure of any early stages would result in the subsequent stages being skipped and being marked as having failed. On the other hand, some natural language tasks, such as ensuring the `train` job specifies a directory for artefact creation and specifying that the `eval` and `model_test` jobs require the `train` job, were generally unproblematic.

10.6.3 Evaluation Scenarios

For the individual projects and scenarios, GPT-4o exhibited the lowest or equal-lowest error rate in most cases. Even in more complex cases, GPT-4o maintained low error rates, indicating superior performance and reliability in generating MLOps pipelines compared to the other LLMs.

10.6.4 Common Errors Committed by the LLMs

We identified and categorised the most common errors made by LLMs across their corresponding task categories. It should be noted that, due to the unpredictable nature of LLMs, these errors cannot be foreseen when first implementing a solution such as ours. The errors only become apparent when running the tool and analysing what went wrong. These errors were straightforward to resolve in a single iteration – see Section 10.7 – by introducing simple improvements to our low-code specification and conversation templates based on our findings below, leading to even lower error rates.

Generation Errors

***GPI*: The generated script uses a correct Python image.** Occasionally, either no Python image or an incorrect one was configured. Sometimes, a Python image was specified only for specific stages. These mistakes resulted in runtime errors during training or evaluation because the `python` command was not found or the `pip` command could not be run due to the Docker environment being externally managed. These errors affected Qwen2.5 Coder when running *SHO2*and Llama 3.3 for *KAZ2* and *SNA2*. Please note that we did not always specify the use of a Python image, let alone a specific version, so a simple fix would be to always specify a particular version, which should have global scope.

***GAR*: All Python requirements and the necessary Linux packages are installed for all required stages correctly.** On one occasion (Code Llama and *SHO2*), the LLM used the `apt` command instead of `pip` to install the Python prerequisites. It installed the wrong version of `protobuf`, causing runtime errors during pipeline execution. The solution is instructing the LLM to install Linux requirements with `apt` and Python requirements with `pip`.

***GEN*: All specified environment variables are set correctly.** LLMs sometimes set environment variables using the `export` command rather than a global `variables:` section

of the pipeline configuration, or they would set environments only within the scope of a specific stage. This error arose with Code Llama (*KAZ3* and *SHO2*), Qwen2.5 (*SNA3*) and DeepSeek (*KAZ3*). It resulted in pipeline stages (e.g. `deploy`) failing due to these environment variables being undefined for those stages. A way to solve this problem is to specify that all environment variables should be set globally in the `variables:` section of the pipeline configuration.

***GDA*: All Docker artefacts are created correctly.** Sometimes, the generated pipeline configuration did not specify the model artefact for the `train` stage. The solution is to state that the model should be provided for a given stage in the `artifacts:` section of the stage. Errors were mainly found with GPT-3.5 (*SNA2*), Code Llama (*KAZ2*, *SNA2* and *SHO2*) and DeepSeek (*SHO2*).

Natural Language Errors

***NET* and *NMT*: The `eval` and `model_test` jobs need the `train` job and its artefacts.** In several cases, either the `eval` (*NET*) or `model_test` (*NMT*) stage did not specify the need for the `train` stage. For *NET*, the issue arose with every model for *SHO1*, whereas for *NMT*, mainly GPT-3.5 was affected (*KAZ3*, *SNA3* and *SHO3*). This omission was not a problem in practice, since artefacts from prior stages were passed to the next stage by default in GitLab.

***NSK*: An SSH key setup script is created correctly for reaching the deployment server.** This problem occurred frequently and caused runtime errors during the `deploy` stage when using secure copy. Worst affected were GPT-3.5 (*KAZ3*), Qwen2.5 Coder (*SHO3*) and DeepSeek (*KAZ3* and *SHO3*). The LLMs tried various ways to create the SSH key, including storing it in an authentication agent instead of a file, not creating the `.ssh` directory, trying to write the private key to the `/root` directory, not setting correct permissions on the `~/.ssh/id_rsa` file, not adding the host to `known_keys` or completely omitting SSH setup. A solution would be to provide the explicit SSH script (optionally as a self-contained file) for the pipeline stage to execute, since the steps would rarely change once initially defined.

***NSC*: Secure copy is used to perform the deployment with the correct user/key, host and directory.** Another frequent issue was deploying the model using the `scp` (secure copy) command. Sometimes, LLMs would use `rsync` instead, or use an incorrect target path or hostname. The LLMs sometimes did not ensure the target path existed on the server or create it if necessary. The secure copy procedure would also fail if the SSH setup was absent or had failed. Similar to *NSK*, the solution would be to provide the steps as an executable script or be more specific about the values and commands to use. The most impacted were GPT-3.5, with a 100% failure rate for each of *KAZ3*, *SNA3* and *SHO3*, Qwen2.5 (*SNA3*), Code Llama (*SNA3*) and DeepSeek (*SHO3*).

***NDA*: The `deploy` stage provides Docker results as an artefact.** Infrequently, the Dockerfile instead of the generated Docker image would be configured as the artefact for the `deploy` stage. Sometimes, the generated pipeline configuration did not specify any Docker image artefact. The solution to both problems is to specify that the generated Docker image should be provided in that stage in the `artifacts:` section of the stage. Most LLMs were affected, in particular GPT-3.5 (*SNA2*), Qwen2.5 Coder (*KAZ2* and *SNA2*), Llama 3.3 (*SNA2*), Code Llama (*KAZ2*, *SNA2* and *SHO2*) and DeepSeek (*KAZ2*, *SNA2* and *SHO2*).

***NDI*: Docker in Docker (DIND) is realised, and the correct image is selected.** A frequent mistake was using the wrong DIND image or omitting DIND altogether. The result during pipeline runtime was that specific commands were not found (e.g. `pip` and `docker`). Again, most LLMs were impacted: Qwen2.5 (*KAZ2*, *SNA2* and *SHO2*), Qwen2.5 Coder (*SNA2*), Llama 3.3 (with a 100% error rate for *KAZ2*, *SNA2* and *SHO2*), Code Llama (*SNA2*) and DeepSeek (*KAZ2* and *SNA2*). On one occasion, a specified version and entry point prevented the Docker daemon from starting (GPT-3.5 running *KAZ2*). The solution is to be explicit about which image version to use, and to specify that `docker:dind` should be used under `service:` in the `deploy` stage.

Runtime Errors

***RUT*: Unit tests ran successfully.** Sometimes, the LLM erroneously assumed that unit tests are executed via `pytest` (GPT-4o for *SHO3*, Qwen2.5 for *KAZ3* and *SHO3*, Code Llama for *KAZ3* and DeepSeek for *SHO3*). On another occasion, the LLM would get unit tests, model tests and model evaluation mixed up and attempt to run scripts meant for a different stage (GPT-3.5 for *SHO3*). These issues can be avoided by providing the LLM with a clear example of how to run the tests.

***RTR*: Model training ran successfully.** Other than problems resulting from no Python image being specified during generation, there were no problems with the model training stage. The solution is to specify the precise Python image to use, as for *GPI*.

***RTE*: Model tests ran successfully.** Like *RTR*, errors were mainly caused by issues with the Python image (see *GPI*). More precise instructions to the LLM would help in these cases.

***REV*: Model evaluation ran successfully.** The most frequently encountered issue with this task was incorrect environment variable settings, which caused the model from the `train` stage to not be found. The solution is the same as the one described for *GEN*, i.e. to set the environment variables globally.

***RDP*: Model deployment ran successfully.** In addition to the reasons for failure described earlier, it sometimes seemed that the LLM did not understand what "deployment"

means in the pipeline context. As a result, the pipeline would be configured to push the Docker image to *registry.gitlab.com* (Qwen2.5 Coder and Code Llama for *KAZ2* and DeepSeek for *SNA2*) or would assume that the model was destined for Kubernetes (DeepSeek for *SHO2*) and would use the `kaniko-builder` command. Another example of the LLM misinterpreting the specification led to unexpected commands, such as changing the directory (with the command `cd ..`), e.g. with Qwen2.5 running *SHO2*.

10.7 Improvements and Reevaluation

Based on our findings in Section 10.6.4, which examine common errors committed by the LLMs, we made several improvements to the low-code natural language template and the prompt text used in our conversation template, with a view to running another evaluation after implementing the changes. We describe the changes in this section. Later in this section, we report on the improved results from the reevaluation.

10.7.1 Improvements to the Low-Code, Natural-Language Template

Our overall approach remains as in Section 10.4.3, with some enhancements. These changes represent a maturation of the CI/CD specification template, transitioning from a basic functional setup to a more professional, maintainable and standardised approach that more closely follows GitLab CI best practices whilst ensuring consistent execution environments, artefact management, resource allocation, more precise documentation and more reliable deployment, addressing our prior findings.

Table 10.11 summarises the changes we made to the low-code template, which are discussed below. The modifications specify enhanced script usage instructions to the LLM by providing more precise documentation for each command. The improved specification template replaces ambiguous single-script invocations, such as `python snake_rl.py ?train|eval|unit_test|model_test?` with four explicitly labelled commands for `train`, `eval`, `unit_test` and `model_test`, each invoking a dedicated script. This change makes the usage of each command immediately apparent. The file paths are also made more explicit by adding `./` prefixes, which follows better scripting practices and removes any ambiguity about file locations.

An essential addition in the improved version is the Python Docker image specification, for instance `python:3.8-bullseye`, for all pipeline stages. This addition ensures consistent runtime environments across the entire CI/CD pipeline, eliminating potential issues caused by environment variations. The original version left the runtime environment unspecified, which led to inconsistent behaviour across different pipeline stage runs.

The enhanced solution introduces structured dependency management by requiring Python packages to be installed in a dedicated `before_script` section using `pip`. This approach separates dependency installation from the main execution logic, making the pipeline more maintainable and following GitLab CI best practices. The original version did not specify where or how these requirements should be installed.

The improved implementation requires all stages to use the `gpu-runner` tag if specified,

Table 10.11: Comparison of Changes to the Low-Code Template

Aspect	***Original***	***Change***
Usage	Unlabelled commands.	Labelled commands with file paths.
Runtime Environment	Unspecified Python image.	Named Python image for all stages.
Dependency Management	Unclear for which stages dependencies should be installed and when.	Introduction of a `before_script` to specify dependencies separate from execution logic.
Dependency Management	Program for installing dependencies unspecified.	Specify `pip` for dependency installation.
GitLab Runners	No GitLab runners specified.	Introduction of optional runner tags.
Terminology	Inconsistent.	Correct use of `job` and `stage`.
Terminology	"Git**l**ab", "**d**ocker".	"Git**L**ab", "**D**ocker".
Configuration	Scope of environment variables undefined.	Environment variables set globally (for all stages).
Docker-in-Docker (DIND)	Mention using DIND.	Precise Docker-in-Docker (DIND) specification for the `deploy` stage.
Deployment Stage Setup	Python dependencies installed.	Override `before_script` to skip Python dependencies installation.
Deployment Strategy	Deployment to a Docker container provided as an artefact.	Save a Docker image and provide it as an artefact.
Deployment Strategy	Description of the deployment steps.	Deploy steps are outsourced to an external script.
Artefact Management	No explicit instructions.	The model artefact for the `train` stage should be provided in the `artifacts:` section of the `train` stage

ensuring consistent resource allocation across the pipeline. Whilst not specifically addressing output correctness, this change can expedite pipeline execution and move the implementation closer to production readiness.

We also refined our terminology. References to `job` and `stage` now align with proper GitLab CI terminology. Capitalisation of `GitLab` and `Docker` has also been corrected.

The enhanced version emphasises centralised configuration practices by mandating that all environment variables be defined globally in the pipeline's `variables:` section. This practice promotes consistency, simplicity, and easier maintenance compared to stage-specific variable definitions.

The new implementation provides a detailed Docker-in-Docker (DIND) configuration for the deploy stage. Whilst the original version mentioned using DIND generically,

the enhanced version specifies exact requirements, i.e. using `docker:latest` as the base image and setting `docker:dind` as a service. We also specify an empty `before_script:` to avoid conflicts with Docker operations. This level of specificity prevents common DIND configuration issues.

Deployment has been refined from container deployment to image artefact creation. The original template described `deployment to a Docker container that is provided as an artifact`, which was somewhat ambiguous. The enhanced solution clarifies that the deploy stage should `save a Docker image and provide it as an artifact`, making the deployment output more explicit and reusable.

The deployment approach has also undergone a fundamental transformation, from the LLM having to guess the deployment commands to use based on ambiguous instructions, to script-based execution. Typically, the deployment script would be provided by the operations infrastructure team. The original implementation specified SSH deployment tasks, including server, user credentials, SSH key management and file transfer operations, but not the commands. The improved solution outsources these operations to an external `deploy.sh` script that must be made executable and executed, simplifying the pipeline configuration whilst improving modularity and maintainability.

The enhanced tool makes artefact configuration more explicit and comprehensive. Both configurations produce model artefacts in `models/`, but the enhanced template requires defining these outputs in the `artifacts:` section of the train stage. This declaration ensures that downstream processes or future jobs can reliably locate and use the trained model files.

10.7.2 Improvements to the Prompting Method

We made several changes to the prompting method. The overall structure of the conversation template and the prompting method remain the same as described in Section 10.4.6, but we streamlined the content to achieve more accurate output. The key differences between the original and new implementations are listed in Table 10.12 and described below.

The new implementation represents a more mature, production-ready version with more accurate terminology to guide the chosen LLM in generating more correct output. Overall, our changes represent a maturation of the tool's capabilities: it is more targeted in its application and speaks with a clearer, more confident tone whilst maintaining the structure of the original conversation template.

The revised question prompt shifts the focus from broad "Python machine-learning scripts" to the narrower domain of "Python reinforcement learning scripts", signalling a more precise use case. This update clarifies the nature of the task and aligns it with RL projects rather than generic ML tasks. This text can, of course, be modified depending on the target project type (ML vs RL). By honing in on RL, the template better reflects that subfield's specialised requirements and typical conventions.

We also refined the system and instruction messages to convey a more authoritative, professional voice. What was previously phrased as a `helpful code assistant that can teach a junior developer how to code` in the system message is now an `expert`

Table 10.12: Comparison of Changes to the Conversation Text

Aspect	***Original***	***Change***
Question Wording	Write a GitLab CI/CD yml file for **Python machine-learning scripts**.	Write a GitLab CI/CD YAML file for **Python reinforcement learning scripts**.
Sample Configuration and Output	No GitLab runner `tags` section in any jobs.	Jobs can include a `tags` section specifying a GitLab runner.
System Message Wording	"You are **a helpful code assistant that can teach a junior developer how to code**."	"You are an **expert programmer**."
Instruction Wording	"Don't explain the **code, just generate the code block itself**."	"Don't explain the **file; just generate it**."
Terminology & Capitalisation	"Git**l**ab", "**d**ocker", "**yml**".	"Git**L**ab", "**D**ocker", "**YAML**".

`programmer`. The instruction `Don't explain the code, just generate the code block itself` in the instruction message has been streamlined to `Don't explain the file; just generate it`. These adjustments shorten the phrasing and elevate the assistant's perceived expertise, increasing the likelihood of delivering concise, expert-level assistance.

Terminology in the system messages and specifications has been standardised for consistency and clarity. References to `Gitlab` have been updated to `GitLab` in keeping with the proper product name, and the capitalisation of `Docker` has been corrected. The description of the CI/CD configuration has also shifted from `yml file` to `YAML file` where appropriate, reflecting the more formal nomenclature used in professional documentation and clarifying the expected markup format. These changes aim to increase the correctness of the LLMs' generated output. The terms were also updated in the rest of the codebase where appropriate.

A significant enhancement in the new version is the introduction of `gpu-runner` tags, if desired, across all job stages. In the updated sample specification, a `gpu-runner` tag is specified for every stage except any hypothetical deploy stage, and each job block in the generated YAML now includes a `tags` section listing `gpu-runner`. This change can, of course, be customised to the user's infrastructure. For instance, the `tags` specification may be omitted entirely, the runner may be named differently than `gpu-runner`, and the stages for which the tags are specified or omitted may be modified. This modification, whilst not targeting LLM correctness specifically, ensures that jobs are executed on runners equipped with GPU resources, which is critical for compute-intensive RL training and evaluation in production.

10.7.3 Reevaluation

We ran the refined implementation on the best-performing LLM from the first evaluation round, GPT-4o, and evaluated it as previously described in Section 10.5. The results are summarised in Table 10.13. In short, we achieved perfect results for this LLM and the new implementation. GPT-4o had an average error of 0 across all combinations of evaluation variables, projects and scenarios. Based on this success, we did not see the need to continue evaluating the remaining LLMs.

Table 10.13: GPT-4o (Version 2024-08-06) Improved Results: Error Rates

	KAZ1	***KAZ2***	***KAZ3***	***SNA1***	***SNA2***	***SNA3***	***SHO1***	***SHO2***	***SHO3***	***Average Error***
GPI	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
GAS	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
GSO	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
GAR	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
GEN	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
GMS	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
GCD	NA	0.00	NA	NA	0.00	NA	NA	0.00	NA	0.00
GDA	NA	0.00	NA	NA	0.00	NA	NA	0.00	NA	0.00
NTA	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
NET	0.00	0.00	0.00	NA	0.00	0.00	0.00	0.00	0.00	0.00
NMT	NA	NA	0.00	NA	NA	0.00	NA	NA	0.00	0.00
NSK	NA	NA	0.00	NA	NA	0.00	NA	NA	0.00	0.00
NSC	NA	NA	0.00	NA	NA	0.00	NA	NA	0.00	0.00
NDA	NA	0.00	NA	NA	0.00	NA	NA	0.00	NA	0.00
NDI	NA	0.00	NA	NA	0.00	NA	NA	0.00	NA	0.00
RUT	NA	NA	0.00	NA	NA	0.00	NA	NA	0.00	0.00
RTR	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
RTE	NA	NA	0.00	NA	NA	0.00	NA	NA	0.00	0.00
REV	0.00	0.00	0.00	NA	0.00	0.00	0.00	0.00	0.00	0.00
RDP	NA	0.00	0.00	NA	0.00	0.00	NA	0.00	0.00	0.00
Average Error GS	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Average Error NL	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Average Error RJ	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Average Error	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

10.8 Integration of our Approach in an Industry Setting

This section describes the approach for applying our solution in an industrial CPPS. This integration is currently being developed in collaboration with an industry partner as part of a research and development project for the next generation of intelligent CPPSs. The tool finds particular applicability in scenarios where pipeline configurations must be continuously created and edited at scale, i.e. for many factories. Figure 10.7 depicts a typical CPPS setup.

In Figure 10.7, our tool (**LCF** – Low Code Flow) enables RL engineers, software developers, and MLOps and RLOps practitioners (represented collectively as the **RL Engineer**) to generate GitLab CI/CD pipeline configurations based on our low-code template using an **LLM**. The **RL Engineer** is the human agent within the system architecture and focuses on designing, maintaining and optimising industrial-level RL solutions.

The **Software Engineering Environment** is where the **RL Engineer** performs their primary work activities. This environment comprises key elements which facilitate successful development processes. The **IDE** (Integrated Development Environment) provides the **RL Engineer** with a complete set of development tools tailored to RL development, including debugging support for complex multi-agent systems and visualisation tools to understand agent behaviour. It can simplify the development process and cut the complexity of creating the RL system. The **RL Engineer** may make use of **LLMs**, **chatbots** or integrated AI tools like **Copilot** – **LLM/Chatbot/Copilot** in Figure 10.7. These tools assist the **RL Engineer** with intelligent support such as coding suggestions, auto-documenting and problem-solving. More importantly, for the context of this chapter, the **RL Engineer** can also use our program, Low Code Flow (**LCF**), which utilises an underlying **LLM**.

The **Code Repository** in the **RLOps Environment** provides version control and collaboration infrastructure, ensuring that all RL algorithm implementations, configuration files and system integration code are correctly tracked and managed. This aspect matters because during RL development, extensive experimentation and iteration are common and robust version control is crucial to ensuring reproducibility. As the **RL Engineer** commits code changes to the repository, the **RLOps CI/CD Pipeline** automatically triggers, demonstrating automated integration between development and operational deployment. The difference with our solution is that **LCF** can create and edit an **RLOps CI/CD Pipeline** and commit it to the **Code Repository**, which also triggers an **RLOps CI/CD Pipeline** run.

The **RLOps CI/CD Pipeline** within the **RLOps Environment**, triggered as described, is the automation engine that coordinates the building, testing, validation and deployment of RL models. It aims to ensure that new model versions are evaluated systematically before deployment and that system reliability is ensured with the possibility of ongoing improvement. The **Model Repository** is a centralised model versioning and storage system that tracks metadata regarding model performance, model training parameters and deployment history. This aspect is essential, as RL models require extensive training and often depend heavily on training conditions, so it is essential that

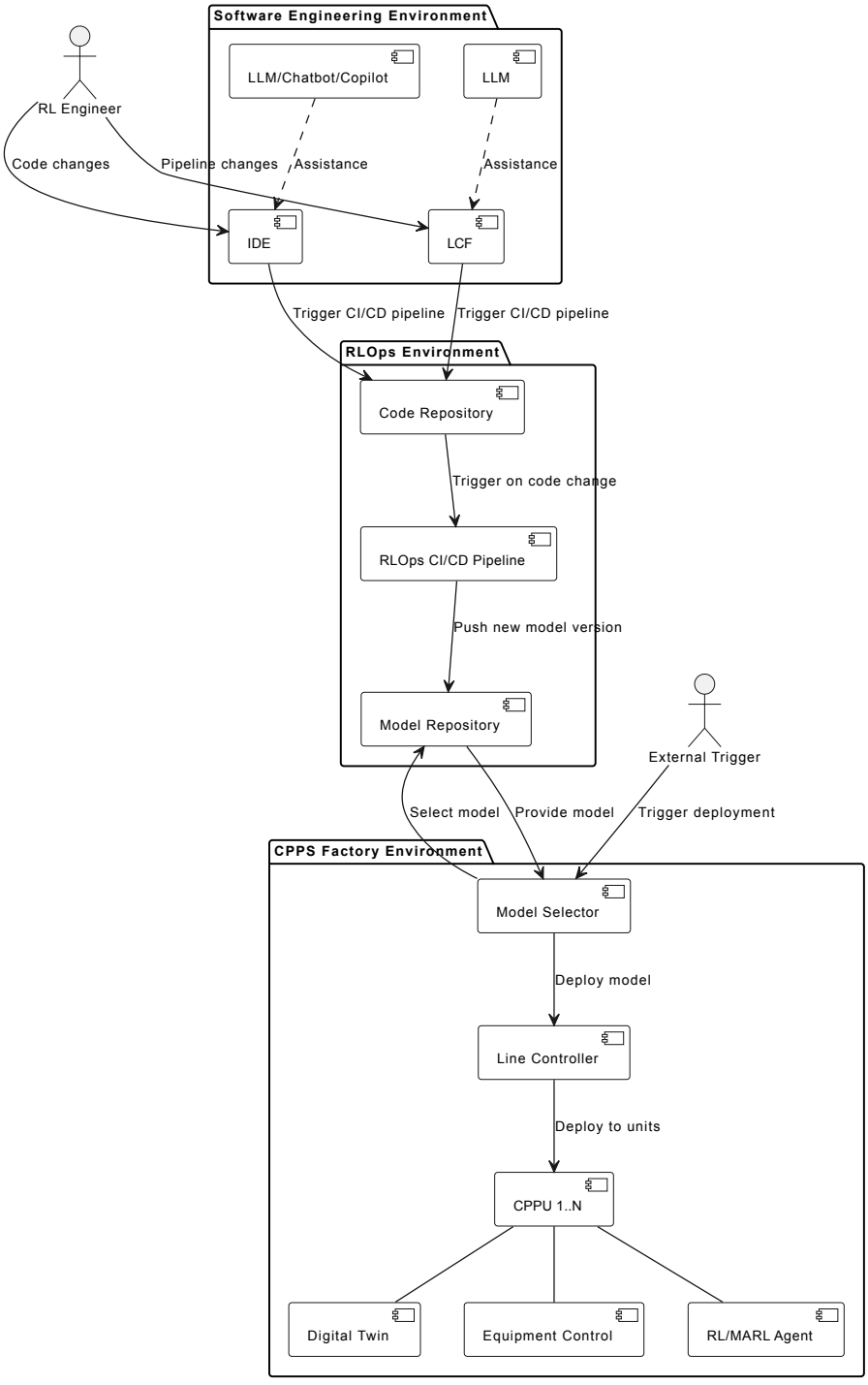

Figure 10.7: Integration of our approach in an industry setting.

production systems correctly version and manage metadata.

The **CPPS Factory Environment** in Figure 10.7 is the physical manufacturing environment where RL algorithms are implemented on real-life production processes. An **External Trigger**, such as changes in real-world model factory performance according to predefined performance metrics, can trigger a new deployment within the **CPPS Factory Environment**. The **Model Selector** may be used to deliver models to production factories. It offers intelligent model selection features that dynamically select suitable models from the **Model Repository** based on existing operational conditions, production needs or performance indicators. It enables adaptive manufacturing systems to adjust their intelligence automatically in response to changing conditions without requiring human intervention. The **Line Controller** is the main orchestration component of manufacturing processes that organises the work of several **CPPUs** and coordinates the operations, deploying models to any number of such **CPPUs**, each of which may have a different RL strategy at its core and be associated with robots with varying capabilities and competencies. **CPPU**s are extendable manufacturing units, which are physical equipment with cyber functionality. These units are necessary because they are the real locations where models make decisions that affect physical production.

The **Digital Twin** is a real-time virtual representation of physical manufacturing assets, which can be simulated, predicted and optimised without affecting real production. It is important because it enables **RL/MARL Agents** to experiment with strategies and estimate their outcomes with great certainty before changing the physical world.

Equipment Control directly communicates with actual manufacturing machinery and converts high-level commands issued by the **RL/MARL Agents** into explicit equipment instructions. This element is necessary to ensure the safe and effective physical implementation of AI-generated strategies.

The **RL/MARL Agents** are the heart of AI that can learn and optimise manufacturing processes using either an RL or MARL framework. These agents are the final frontier of the system architecture, where models are deployed to make real-time decisions that directly affect production efficiency, quality and adaptability.

10.9 Discussion

This section discusses our results in the context of the research questions defined in Section 10.1.

RQ$_{5.1}$: *How accurately can a low-code, template-based modular flow approach – leveraging LLMs for generation and validation – produce correct MLOps pipeline configurations for RL?* Our empirical validation confirms that our low-code, template-based approach can generate MLOps pipelines with low error rates, contingent on the specific LLM selected for the task, which is discussed in further detail below under **RQ$_{5.2}$**. Initial average error rates across all evaluation variables, projects and scenarios ranged from 0.09 (GPT-4o) to 0.3 (Code Llama). Regarding task categories, our approach achieved a best error rate of 0 for generation tasks, 0.09 for natural language tasks and 0.26 for runtime tasks, all achieved by GPT-4o.

We also showed that error rates can be reduced by making only minor improvements to the implementation and minor corrections to the pipeline specification, thereby achieving perfect results for GPT-4o. Typically, these minor amendments are carried out by individuals closest to the pipeline's domain: MLOps and RLOps specialists, domain experts, ML and RL engineers, or, in leaner teams, experienced data scientists who wear multiple hats. Since our approach flags errors via the LLMs' validation feedback or through straightforward GitLab failure messages, remedying issues often amounts to little more than updating some text in the low-code template and pipeline specification instances, or being more precise in the conversation template. Ample examples of correcting the errors noted in Section 10.6.4 are documented in Section 10.7.

Such tweaks as those we implemented impose a minimal burden on non-expert users, as the requisite details (for instance, using approved container images) are usually supplied alongside the pipeline templates by the underlying infrastructure team. Errors of this nature typically arise because the user neglected to specify the correct image in the first place. If, however, the LLM itself generates a pipeline configuration that does not succeed at runtime, for instance because it erroneously used an incorrect command to install Python dependencies (like `apt` instead of `pip`), then the user can add natural language text to the pipeline specification file that `Python requirements should be installed using pip` and rerun the tool. This continuous refinement can also help reduce future error rates.

Combined with selected LLMs' abilities to handle natural language inputs, our approach is potentially a good option for users who may not have deep DevOps knowledge, helping make pipeline development more accessible by allowing users to participate in pipeline production without extensive software engineering or DevOps knowledge. Our templates support a wide range of pipeline features. With their rigid structure, they are designed to guide practitioners, making it easy for users to specify their needs in natural language, whilst also being detailed enough for LLMs to produce correct output, reducing the likelihood of a mismatch between expected and generated output.

We see a practical benefit of our approach: it could reduce development time by allowing users to work with templates for pipeline descriptions, thereby decreasing the time spent learning to write and debug GitLab pipeline configurations directly. The templates may enable rapid development, particularly in complex or dynamic settings with many pipelines or continuous pipeline creation or adaptation. Indeed, we plan to test the potential for increased production speed in a user study in future work (see Section 10.11).

Our findings provide concrete contributions. We identified common errors that different LLMs make when generating pipelines, which can guide practitioners in improving communication with LLMs when implementing an approach like ours. A key takeaway is that the limitations of poorly performing LLMs appear to stem from ambiguity or insufficient detail in parts of the pipeline specifications. Being more precise in the template and providing more GitLab-specific knowledge to the LLM can further improve results – we achieved perfect results with GPT-4o after doing so. A further contribution of this study is our comprehensive set of evaluation variables and the resulting error rate metric, which can be used to assess the correctness of any MLOps pipeline or automated approach with minimal modification and could be applied in similar studies or applications.

RQ$_{5.2}$: *How accurately do different LLMs generate MLOps pipeline configurations for RL, and what are their limitations?* Comparative analysis of our initial results demonstrates that OpenAI's GPT-4o model performed best across all task types, with an average error rate of just 0.09 and was the overall best-performing LLM. This result represents high consistency, reliability and accuracy in generating correct outputs. GPT-4o outperformed the other LLMs across most evaluation variables, with lower or equal average error rates in most cases.

Qwen2.5 Coder also performed well, achieving an average error rate of 0.15, with a generation task error rate similar to that of GPT-4o (0.04 and 0 respectively). Qwen2.5 Coder performed much worse than GPT-4o for natural language (0.17 vs 0.09 respectively) and runtime tasks (0.36 vs 0.26 respectively). Many LLMs had no errors for multiple evaluation variables across all projects and scenarios. In particular, most LLMs achieved an error rate of 0 for the generation tasks corresponding to *GAS* ("the generated script contains all required stages"), *GSO* ("the required stages are in the correct order") and *GAR* ("all Python requirements and the necessary Linux packages are installed for all required stages correctly"). GPT-4o consistently achieved perfect results for a broader range of evaluation variables, suggesting users can expect more reliable output from a pipeline built with GPT-4o using our approach. We noted low variability in error rates for GPT-4o, which was much more consistent across projects, scenarios and tasks, and performed remarkably well. All models struggled with the deployment task corresponding to *RDP* ("model deployment ran successfully"), which proved challenging at runtime, and the lowest error rate was only 0.72 (achieved by Qwen2.5 Coder).

GPT-4o demonstrated an average error rate of only 0.09, beating more specialised coding models like Qwen2.5 Coder, Code Llama and DeepSeek R1 (with average errors of 0.15, 0.3 and 0.22 respectively), which is both surprising and indicative of advances in model design and training techniques. Despite their specialisation in code generation, these LLMs could not compare to GPT-4o in terms of consistency and accuracy. Even broader models such as GPT-3.5 Turbo, Qwen2.5 and Llama 3.3 (with average error rates of 0.19, 0.18 and 0.18, respectively) could not match GPT-4o, indicating that the architectural improvements and multimodality of GPT-4o provide a practical benefit for tackling various tasks without sacrificing depth and accuracy. The fact that GPT-4o performed consistently well across projects and scenarios indicates its stability, making it an ideal option for pipelines where output consistency is paramount. This consistency is compounded by GPT-4o's perfect performance in the enhanced version of our tool, even after minimal changes.

Our results raise the question of what specialised LLMs can reasonably be expected to accomplish compared to more general LLMs. It could be expected that a model designed specifically to code – like Code Llama or DeepSeek R1 – would perform well on generation or runtime benchmarks that involved programming tasks. Still, these models performed worse overall on deployment and runtime tasks. The fact that these coding-oriented LLMs underperform shows that specialisation does not necessarily imply lower error rates in all task dimensions. Instead, the balanced and subsequent perfect performance of GPT-4o on natural language generation and execution tasks implies that multimodal training and

inference optimisations can be more reliably integrated to produce an overall performance greater than the sum of its parts. To organisations considering model selection, these results point to the importance of flexibility and robustness over domain specialisation and that even the most accurate coding LLMs may need supplementary validation mechanisms in production, such as the ones we implemented, to counter edge-case failures.

Our comparison shows that LLMs vary in performance, and standard LLMs can produce impressive results. A key observation is that code-specific LLMs should not be automatically assumed to perform better than non-code-specific LLMs in code generation applications such as ours. Combining the strengths of multiple LLMs, fine-tuning and more advanced prompt engineering and error-correction techniques, whilst focusing on the most challenging tasks, may yield even better results. Alternatively, by being more precise in our pipeline specification and conversation template, we improved results even further.

10.10 Threats to Validity

In this section, we address threats to validity as outlined by Wohlin et al. [85].

Internal Validity The correctness of the produced pipelines depends on how comprehensive and precise the templates are. LLMs' capabilities to accurately understand and generate code from template-based and natural language instructions can vary. Incorrectly interpreting templates, particularly natural language inputs, can lead to mistakes or less-than-ideal configurations, which in turn may cause problems in the generated pipelines. In cases where we implemented specific checks for the contexts of the generated configurations, this issue was often considerably reduced.

The way prompts are designed also heavily influences the correctness of LLM-generated pipelines. If prompt phrasing is unintentionally biased towards certain LLMs, some models may perform better simply because they align more closely with the prompt style than their actual capabilities. We sought to mitigate this risk by using carefully formulated prompts, applying few-shot prompting, prompt chaining, error detection and retries to guide the LLMs. We manually inspected LLM output over numerous scenarios, evaluation variables, projects and LLMs. We did not detect any incompatibility or bias that was obviously linked to the prompt style. We refined the prompts during the initial development phase (before beginning the evaluation) until we started receiving reasonable outputs from the LLMs.

External Validity We assess our approach based on a small number of RL cases, which may not apply to other fields or project types. Due to the scarcity of open-source MLOps projects utilising RL, we had to develop the ground truth pipelines for our case studies independently. Although we adhered to industry practices (as documented in the literature [8], [23], [24] and industry projects), we may have overlooked some essential MLOps practices. Differing projects might need considerable modifications to the specification and generation classes. Whilst these factors could limit the immediate use of our methodology, the fundamental approach should remain relevant in similar

situations. The effectiveness of our approach is contingent on the capabilities of the underlying LLMs. Differences in the quality of the model or the inputs given to it can affect the standard of the generated pipelines. We tackled this concern by conducting tests on various cases, evaluation scenarios and LLMs.

Construct Validity The metrics used to assess the generated pipelines may not fully capture all aspects of their performance, and all evaluation metrics carry equal weight in our assessment. In contrast, some factors may be more significant in practice than others. In this case, one could re-evaluate our findings using a set of weighted metrics.

The effectiveness of the natural language-based specification components relies on the accuracy of the user's inputs. Therefore, slight variations in human input can lead to inconsistencies in the generated outputs, potentially affecting the validity of the results. Likewise, differently structured prompts could yield different generated outcomes. We ran our evaluation multiple times for multiple scenarios, projects and LLMs to account for this eventuality.

Conclusion Validity The results of our experiments rely on a small number of examples and cases. A broader and more varied range of case study projects might strengthen the validation of our approach. Any biases present in the training data of the LLMs we used could have affected the generated pipelines. We addressed this concern by conducting tests with multiple LLMs.

10.11 Conclusion and Future Work

In this chapter, we present a novel approach for automating the generation of MLOps pipelines for RL applications, employing a low-code, template-based, modular approach that leverages LLMs to enable users with minimal coding expertise to generate MLOps pipelines for testing, model training and validation and model deployment. Our evaluation indicates low error rates for pipeline generation, natural language comprehension and runtime tasks, with GPT-4o performing best amongst seven LLMs and achieving a perfect average error rate of 0 after simple refinement of our tool. The results indicate that LLMs, when approached with a validation and correction strategy, can foster trustworthiness in software by minimising errors in the generated code and issues with LLM-based code generation.

To further validate our solution, assess its usefulness and inform future enhancements, we plan to conduct a quantitative user study comparing the use of our approach to manual pipeline configuration creation as part of an Industry 4.0 case study in collaboration with domain experts. We suggest that using our proposed system could drastically reduce coding requirements, accelerate pipeline development and bring MLOps closer to data scientists and other domain experts who may lack advanced software engineering or DevOps skills, thereby democratising MLOps and simplifying pipeline creation for practitioners with limited technical know-how.

Other potential directions include fine-tuning LLMs and developing more advanced error detection and correction mechanisms within the pipeline generation process to support a broader range of LLM types. These improvements could target weaker tasks, such as model deployment. Another enhancement could be incorporating a feedback mechanism that learns from deployed pipelines to improve future pipeline generation.

Finally, this study focused on the applicability of our approach and the correctness of our generated pipelines. A future study could examine LLM pipeline generation efficiency, including the time required to generate a pipeline and the number of tokens consumed, as these are also factors in practice.

11 RLOps Pipeline Development with Low-Code and Large Language Models: An Industry 4.0 Case Study

MLOps automates common tasks throughout the entire life cycle of an ML model. In Industry 4.0 environments, CPPSs increasingly utilise RL models to optimise and automate production processes, giving rise to RLOps. However, manually configuring RLOps pipelines requires specialised DevOps knowledge and can limit the potential for automation. We apply a template-based approach for rapidly creating and deploying RLOps pipelines in Industry 4.0 environments that reduces the required programming effort and expertise. Based on the Pipes and Filters pattern, our modular solution utilises LLMs to automate pipeline creation. It enables fully automated execution, including model training, testing and deployment, with built-in quality control ensuring correct configurations. We demonstrate our approach with a case study from an Industry 4.0 platform for smart factories, in which we empirically evaluate seven LLMs across four representative RLOps use cases. This evaluation provides the first systematic comparison of LLM capabilities for generating RLOps pipelines in Industry 4.0. Our method achieved perfect results with OpenAI GPT-4o. Our results demonstrate that our solution, used with a suitable LLM, can reliably generate and execute RLOps pipelines with low error rates, thereby accelerating pipeline development time and democratising RLOps deployment by greatly reducing the need for specialised DevOps knowledge.

11.1 Introduction

MLOps [6], [31], [57] is a set of practices for the automated execution of tasks related to various stages of the ML [23], [24] workflow for supervised and unsupervised learning. Applied to RL [2], MLOps is termed RLOps [7]. Industry 4.0 [10], [12] integrates digital and physical technologies, and this has led to the development of CPPSs [15] which improve efficiency, flexibility and customisation [13]. In CPPSs, RL can be used to train intelligent agents with digital twins [16], [61].

The differences between RLOps in Industry 4.0 and the operations of cloud-based ML systems are significant. In contrast to homogeneous, centrally controlled cloud environments, an Industry 4.0 CPPS imposes distinct constraints that affect pipeline design and deployment. Heterogeneous edge nodes with different capabilities require real-time decision-making, whereas cloud pipelines assume homogeneous connectivity. Manufacturing locations have varied installations that need differentiated pipeline settings, whereas cloud scaling has homogeneous installation instances. Furthermore, RLOps for

Industry 4.0 must interact directly with physical production systems, adhere to industrial safety and communication protocols and integrate with existing automation infrastructure. Current CI/CD applications using LLMs, such as GitHub Copilot or Amazon Q Developer, do not align with Industry 4.0 RLOps because they presuppose homogeneous environments. They also lack domain knowledge of industrial protocols, safety measures and the real-time constraints when RL agents operate on physical systems.

MLOps, RLOps and Industry 4.0 CPPSs place great value on automation. Yet one of the activities central to automation – CI/CD pipeline development – remains largely manual. This activity requires specialised DevOps [21], [22] knowledge, as well as software engineering and ML/RL know-how [31], [60]. It may require considerable manual effort, for instance, when porting ML scripts from computational notebooks like Google Colab or Jupyter to scripts for execution in a CI/CD pipeline. The manual nature of this activity can lead to inefficiencies and scalability challenges, particularly in highly dynamic Industry 4.0 settings, which require numerous rapid and accurate changes [59] to CI/CD pipelines across multiple physical deployment sites.

To address these challenges, we apply and evaluate our approach from prior work [72] in this doctoral thesis towards automating RLOps CI/CD pipeline generation based on low-code [261], [262], template-based RLOps CI/CD pipeline specifications for the RL life cycle. Our solution implements the Pipes and Filters pattern [253], applies few-shot prompting and prompt chaining [254], error detection and retries, and uses LLMs to generate GitLab CI/CD pipeline configuration files, which are automatically deployed, based on the provided RL scripts and a low-code specification file written by the user. Our approach applies equally to ML and RL-based systems. It utilises a low-code, template-driven implementation based on natural language, where practitioners write specifications using a textual generation template as guidance, requiring little programming knowledge.

The modular architecture of our approach, combined with the Pipes and Filters pattern, enables the management of the flow between the generation modules and supporting components. Each module focuses on tasks such as generating CI/CD pipelines and detecting problems with them. Quality assurance is provided, including generation retries, linting, validation and automated error correction, all of which help improve the correctness of the generated pipelines. The natural language-based specification enables non-software engineering experts, such as data scientists and domain specialists, to create pipelines without writing pipeline specifications manually.

This chapter is based on **P$_{10}$**, describes **RC$_{10}$** and addresses **RP$_5$**. Our study aims to answer the following research questions to finish addressing **RQ$_5$**:

RQ$_{5.3}$ *How accurately can a low-code, template-based, modular flow approach using LLMs generate Industry 4.0-specific RLOps pipeline configurations?*

RQ$_{5.4}$ *How do popular LLMs perform in generating Industry 4.0-specific RLOps pipeline configurations, and do they commit or encounter any recurring errors or challenges?*

The study is part of an ongoing collaboration between the University of Vienna and Siemens Aktiengesellschaft Österreich. Siemens provides production automation solutions

and is developing an Industry 4.0 CPPS solution to automate factory manufacturing. We evaluate our approach against a project derived from Siemens' real Industry 4.0 pipelines and typical use cases for RLOps CI/CD pipeline development. A crucial aspect of managing production environments is quickly developing such pipelines at scale. This aspect is crucial because CI/CD pipeline deployment requires rapid pipeline creation and adaptation, and numerous pipelines must be developed across multiple factories.

The evaluation of our approach compares the capabilities of seven popular LLMs. Comparing LLMs serves two essential purposes, rather than simply finding the most suitable model for Siemens' use. Firstly, it helps us answer **RQ$_{5.3}$** regarding the accuracy of our overall approach. Secondly, it aids us in addressing **RQ$_{5.4}$**, which aims to discover how popular LLMs perform when generating RLOps pipeline configurations and the nature of the errors they encounter.

We empirically evaluate and compare the pipelines generated by the LLMs with reference configurations and their execution results, considering the resulting error rates. Our work offers two contributions which fill the gap in automated RLOps configuration development: (1) with our natural language, template-based specification approach, a modular architecture is presented where LLMs are used for automating pipeline generation for Industry 4.0 for the first time; (2) we devise evaluation scenarios and variables suited to an Industry 4.0 CPPS to rigorously evaluate the quality of the pipeline generation in an Industry 4.0 context using different LLMs.

The rest of the chapter is organised as follows: Section 11.2 highlights related work and Section 11.3 discusses our approach. Our evaluation via an industry case study is detailed in Section 11.4 and Section 11.5 discusses our findings. Threats to validity are considered in Section 11.6 and Section 11.7 concludes and discusses future work.

11.2 Related Work

MLOps has been widely adopted because it automates ML model deployment; however, it is still fraught with challenges. Zhou et al. [31] recognise the need for MLOps to unify ML system development and address ML model deployment challenges. Sharma et al. [259] study options for implementing ML techniques within CI/CD pipelines, but do not provide a complete solution. Kumara et al. [258] review the industry literature and present a reference architecture for MLOps that highlights the challenges in selecting tools and designing strategies for MLOps environments.

Li et al. [7] study RL applied to open radio access networks (O-RAN). They note the differences between ML and RL and categorise ML/RL model life-cycle challenges, whilst suggesting best practices for RLOps. Del Real Torres et al. [146] review deep RL approaches for smart manufacturing in Industry 4.0 and 5.0 [147], highlighting the importance of edge devices and CPPSs. Kegyes et al. [148] conduct a systematic literature review of RL applications and methods for Industry 4.0, providing a reference.

Low-code development methods have recently gained popularity [261], [262]. Hirzel [263] discusses how low-code programming reduces dependence on programming languages such as C, Java and Python, instead relying on natural and visual languages. Bruhin et

al. [265] consider the connection between LCDPs and generative AI and how generative AI can transform the potential of LCDPs.

LLMs are often used for code generation tasks. Jin et al. [266] systematically assess LLM code generation challenges and suggest guidelines to enhance LLM-based code generation system reliability, robustness and usability. Jiang et al. [267] present a survey on LLM advancements for code generation tasks and develop a taxonomy for recent developments. Li and Murr [268] study how current GPT-4 models generate synthesised programs and state that GPT-4 model variants can effectively solve HumanEval [269] problems.

Our approach complements these studies and our prior work [72] by unifying various related topics, including MLOps, RLOps, Industry 4.0, CI/CD and utilising LLMs for code generation. In doing so, we aim to support practitioners in generating RLOps pipeline configurations for Industry 4.0, thus enhancing this (usually manual) step of the RLOps process.

To our knowledge, no prior work has addressed this topic in a unified manner, and this is the first empirical study that systematically assesses the quality of automatically generated CI/CD pipelines for RLOps in an Industry 4.0 case study using LLMs. Although the literature covers MLOps automation for the cloud and the ability of LLMs to produce general code, no previous studies have discussed the specific limitations of heterogeneous industrial deployments at scale that require differentiated pipeline configurations to be quickly generated and deployed across numerous physical factories. Furthermore, our empirical comparison of seven LLMs corroborates our findings from prior work [72] and provides evidence that challenges existing assumptions about code-specialised versus general-purpose models.

11.3 Approach

11.3.1 Research Process

Our research method is based on case study research in software engineering [288]. We examined the application of our proposed approach, described throughout this section, in a real-world context to gain insights into its practical utility and effectiveness. By focusing on a project with Siemens, we can identify the challenges and benefits of implementing our tool in a realistic RL pipeline and application tailored to their needs, including an empirical evaluation of our approach.

Our solution is being developed in collaboration with Siemens as part of a research and development project for the next generation of intelligent CPPSs [13], [14], to integrate it into their workflow. It applies particularly to scenarios where pipeline configurations must be continuously created and edited at scale, i.e. for many factories. To highlight its use, the intended workflow, i.e. how Siemens should use our tool, is discussed in Section 11.3.5.

We followed a systematic, structured methodology, which is depicted in Figure 11.1. We first conducted preliminary, informal experimentation (**Preliminary Experimentation**) in which we provided LLMs with descriptive natural language prompts to generate RLOps pipeline configurations.

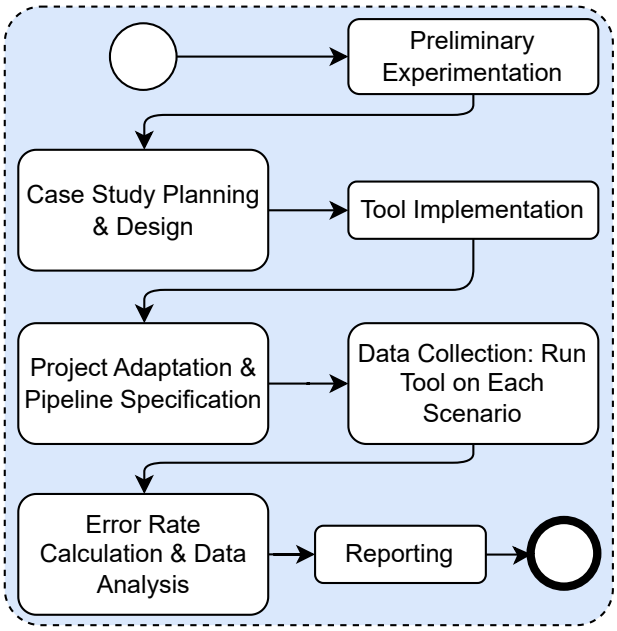

Figure 11.1: Our research process.

We then planned and designed our case study (**Case Study Planning & Design**), as described in Sections 11.4.1, 11.4.2, 11.4.3, 11.4.4 and 11.4.5.

Next, we implemented our tool (**Tool Implementation**) as described in Sections 11.3.2, 11.3.3 and 11.3.4. This step also involved incorporating the improvements we implemented in our prior work [72].

In **Project Adaptation & Pipeline Specification**, we adapted an existing RLOps project and pipeline provided by Siemens to remove dependencies on external components that were not necessary for evaluating our approach. We wrote reference pipeline configuration files for four typical use cases, and wrote pipeline specification files as detailed in Section 11.3.2.

Once we had prepared the files described in the previous step, we ran our implemented approach on the pipeline specifications for each use case, using the seven LLMs (**Data Collection: Run Tool on Each Scenario**) and recorded the outputs. We opted to perform three runs, as this provided a balance between error discovery (we found that by the third iteration errors began to recur) and cost (computing expenses).

For the empirical evaluation, we calculated the error rates (**Error Rate Calculation & Data Analysis**) of the LLM-generated output to the reference pipeline configurations and specifications for each scenario, as well as the expected behaviour of the executed pipelines. More details on the error rate calculation are found in Section 11.4.5, where we discuss our evaluation. We report our results in Section 11.4.6 (**Reporting**).

11.3.2 Template-Based Natural Language Low-Code Specifications

Our approach uses a structured and template-based low-code approach for RLOps pipeline specification, utilising LLMs. All pipeline details are specified in a natural language-based template and serve as a guide for the generation process. Including the information at the level of detail indicated in the template helps reduce errors and improve the accuracy of the generated results. Furthermore, any additional natural language project or pipeline-specific instructions can be included or appended to a specification as necessary on a case-by-case basis. A specification that we pass to the LLM as part of our prompting method (described in Section 11.3.3) is found in Listing 11.1.

```
Write a GitLab CI/CD yml file for Python reinforcement learning
    scripts.
Do not use Docker inside the GitLab CI/CD pipeline. Use the following
     context:

Usage:
train: python ./rl_script.py --train --steps $TRAIN_STEPS
eval: python ./rl_script.py --eval

Required Files:
rl_script.py

Requirements:
pettingzoo[butterfly]>=1.24.0
stable-baselines3>=2.0.0
supersuit>=3.9.0

Python Docker image to use globally (for all stages) under "image:":
python:3.9-bullseye

Environment Variables:
MODEL_PATH "./models"
TRAIN_STEPS 1000

GitLab CI/CD pipeline stages:
train, eval

GitLab CI/CD pipeline generates artifacts:
Yes, it creates artifacts for the models in the train stage

GitLab CI/CD pipeline generates Docker images:
No

Python requirements should be installed using:
pip

# Stage Settings
GitLab CI stage requirements:
```

```
-  train stage creates artifacts at location MODEL_PATH
-  eval stage needs train job and its artifacts

Details of the train stage:
The model artifact for the "train" stage should be provided in the "
    artifacts:" section of the "train" stage.

Tag to use in all stages (except for "deploy", if that stage is present):
gpu-runner
```

Listing 11.1: An instruction and example low-code specification provided to the LLM.

Further examples of specification files that adhere to the template are in our replication package [289] in the `spec.txt` files in the subdirectories of `rl_dataset/evaluation_scenarios/`.

11.3.3 Prompting Method

We use few-shot prompting [254] for each pipeline configuration generation, error detection and validation step. This method includes examples within the prompt to help the LLM generate more accurate results. When retries are required due to detected errors, we include the conversation history and any detected issues so the LLM can improve its output. This technique, known as prompt chaining [254], helps ensure that each step of the generation flow is correctly completed.

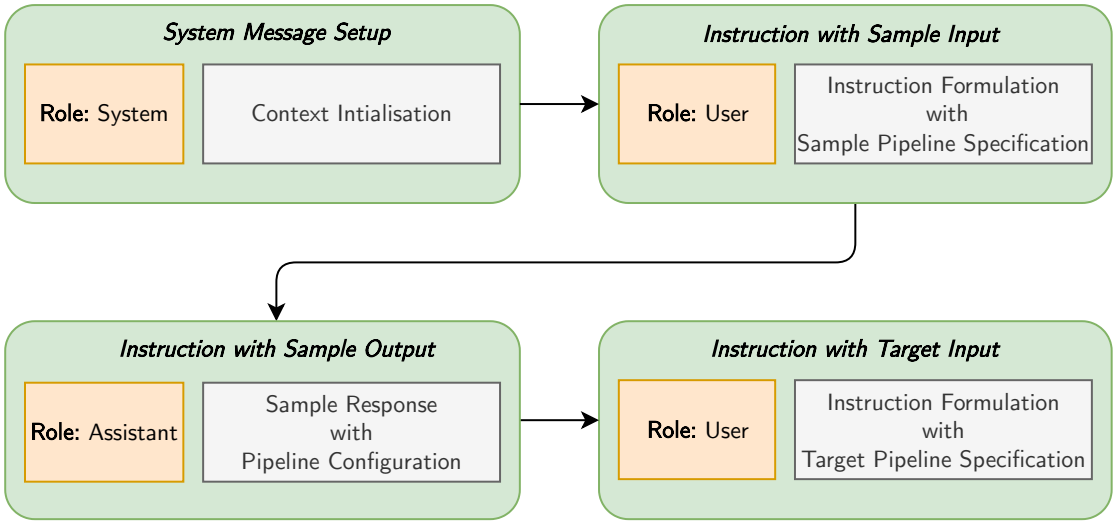

Figure 11.2: Our approach's conversation template.

Figure 11.2 shows how our tool interacts with an LLM to generate a CI/CD pipeline configuration based on a low-code specification. Our tool first creates a system message (**System Message Setup** in Figure 11.2) instructing the LLM to create a GitLab CI/CD YAML file, as in Listing 11.2.

```
You are an expert programmer. Your language of choice is Python and
    you want to create a GitLab CI/CD YAML pipeline configuration file
     (".gitlab-ci.yml") for your project.
Don't explain the file; just generate it.
```

Listing 11.2: Context initialisation provided to the LLM.

It then formulates a specific instruction (**Instruction with Sample Input**) and provides the example pipeline specification already shown in Section 11.3.2. Our tool then provides the LLM with a sample response to the specification (**Instruction with Sample Output**), containing a corresponding output GitLab CI/CD pipeline configuration as in Listing 11.3.

```
default:
  image: python:3.9-bullseye
  before_script:
    - pip install pettingzoo[butterfly]>=1.24.0 stable-baselines3
    >=2.0.0 supersuit>=3.9.0

stages:
  - train
  - eval

variables:
  MODEL_PATH: ./models
  TRAIN_STEPS: 10000

train:
  stage: train
  tags:
    - gpu-runner
  script:
    - mkdir -p $MODEL_PATH
    - python ./rl_script.py --train --steps $TRAIN_STEPS
  artifacts:
    paths:
      - $MODEL_PATH

eval:
  stage: eval
  tags:
    - gpu-runner
  script:
    - python ./rl_script.py --eval
  needs:
    - job: train
      artifacts: true
```

Listing 11.3: An example CI/CD pipeline specification provided to the LLM.

The program then assembles the conversation by combining the system message, the query, the example pipeline specification, the example GitLab CI/CD pipeline configuration file response and an actual pipeline specification from which to generate a GitLab CI/CD pipeline configuration file (**Instruction with Target Input**). If errors or warnings occur, the conversation with the LLM continues, attempting to correct the mistakes by retrying a configurable number of times before selecting the response with the fewest errors or warnings. Please note that our template is extensible, and that real-world specifications, and the specifications used in our case study, are typically more complex than the examples shown in Listings 11.1 and 11.3.

11.3.4 Pipeline Generation Architecture

Our tool consists of multiple components with dedicated responsibilities. **Generator components** include the central elements that execute the generation flows using an LLM. **LLM components** abstract the functionalities of specific LLMs and facilitate the generation of RLOps pipeline configurations based on provided specifications. **Conversation components** manage interactions with the LLM by abstracting conversation instances and maintaining context and interaction state. A **data transfer object** [285] is passed through generation steps in the Pipes and Filters implementation. It contains a generation report that is built as the generation and validation progress. The sequence of operations, such as generation, detection and validation, can be executed sequentially or in parallel. **Detection components** can, for instance, check for errors in YAML syntax, the stage order of generated pipelines, whether a generated pipeline configuration contains code blocks or whether a pipeline stage generates artefacts. A **validation component** might use the GitLab API to check whether the generated pipeline configuration is valid. **Supporting components** are used, for example, to commit the generated pipeline and its project to GitLab.

The execution flow is defined using a simple Python DSL, and an example is given in Listing 11.4. The DSL implements the Pipes and Filters pattern and specifies the generation, detection and validation steps. The pipe definition determines the sequence of events and connects the filters, which do the work. This structured generation flow, with focused steps, contextual feedback, validation and retries, helps address accuracy challenges such as AI hallucinations, which are common with LLMs [272], [273], [274].

```
LCF(components=[
    GenerationRetry(
      GenerateGitlabCIYmlFileFromSpecFile,
        ParallelSuccessfulIfAllSuccessful(
          [
            GitlabCINoLinterIssuesDetection,
            GitlabCIArtifactsCreatedDetection,
            GitlabCICheckStageOrderFromSpecFile
          ]
        )
      ),
```

```
    GitlabCommitGeneratedProject])
```

Listing 11.4: An example use of the Python DSL.

In the example in Listing 11.4, we use retry logic (`GenerationRetry`) to generate a GitLab CI/CD YAML file from a specification file (`GenerateGitlabCIYmlFileFromSpecFile`). We then run error detection measures in parallel (`ParallelSuccessfulIfAllSuccessful`): linting the generated GitLab pipeline configuration file (`GitlabCINoLinterIssuesDetection`), verifying that the pipeline configuration creates artefacts (`GitlabCIArtifactsCreatedDetection`), and checking that the pipeline has the correct order of stages (`GitlabCICheckStageOrderFromSpecFile`). We then commit and push the generated pipeline configuration, along with its project, to GitLab (`GitlabCommitGeneratedProject`) for execution.

11.3.5 Integration and Workflow

Figure 11.3 shows how a practitioner can use our tool.

In the first step (**Write Initial Pipeline Specification**), the practitioner writes an initial version of the pipeline specification for their RL project, based on the template shown in Section 11.3.2. The user then runs our tool on their specification file and project (**Run Tool on Project to Generate and Deploy Pipeline**). Our tool uses an LLM to generate the GitLab CI/CD pipeline configuration file from the specification, which is then automatically committed to a GitLab repository. As the pipeline is running (**Check Pipeline Deployment and Execution**), the practitioner can keep an eye on the pipeline's execution status (i.e. that it deployed and executed successfully) and, if necessary, note any errors that occur. Simultaneously, the practitioner can inspect the generated pipeline configuration (**Manual Inspection of Generated Pipeline Configuration**) to verify that the pipeline configuration file is acceptable, i.e. that there are no obvious mistakes. Suppose an error is apparent in the generated pipeline configuration or its execution. In that case, the user can make corrections to the pipeline specification (**Minor Corrections to Pipeline Specification if Necessary**) and rerun the tool to generate a new pipeline configuration if required. Any corrections would depend on the specific error encountered. For instance, if an environment variable was not set or required packages were not fully specified, these would need to be added at this stage. Thus, each refined specification version would be expected to produce more accurate output from the LLM over several iterations.

The resulting workflow thus features an iterative feedback loop, allowing the user to continuously refine the pipeline specification. Human-in-the-loop manual inspection and approval help maintain accountability of the human operator. In practice, it is unlikely that many iterations would be needed; in Section 11.4.6, we show results that suggest that, at least for this industry case study, one of the LLMs yielded perfect results, i.e. it required no corrections to its output at all.

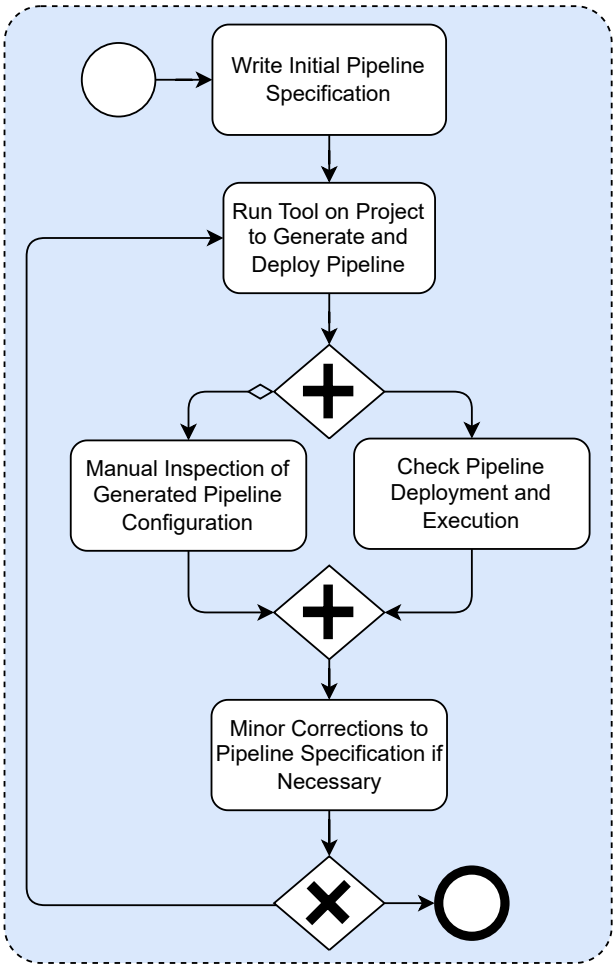

Figure 11.3: Detailed workflow steps for a user of the tool.

11.4 Industry Case Study

11.4.1 Industry Case Study Description

This work is part of a long-term collaboration between the authors and Siemens Aktiengesellschaft Österreich, which provides solutions for Industry 4.0 CPPSs to automate factory production. An essential requirement in industrial applications is the creation and evolution of such pipelines at scale. This requirement is imperative for two reasons. Firstly, CI/CD deployment requires rapid updates, including changes to the pipeline. Secondly, numerous pipelines must be created and developed across projects for different factories.

We applied our approach to Siemens' project using their existing repositories and pipelines and tested it against common use cases for RLOps pipeline development. Our evaluation is based on a project adapted from the industry case that captures the most essential and relevant RL-specific aspects on which our tool operates. The full implement-

ation, the pipelines studied, and our results are available in our replication package [289]. This adaptation excludes confidential implementation details, as we cannot share the industry case's proprietary artefacts due to intellectual property rights and confidentiality agreements. Our solution focuses on the pipeline structure, including stages and tasks, rather than on the details of the RL implementation or on integration with systems that are not directly relevant to RLOps activities. The result is a pipeline specification that encapsulates the desired pipeline structure and activities, with the minimal RL implementation required to test our solution without compromising confidentiality.

The case study project comprises components that train RL models and a production simulator that optimises factory production in a decentralised yet collaborative manner between CPPUs, whilst optimising key performance indicators. At the heart of our case study is a custom environment defined in `factory_robot_env.py`, which models the environment in which a factory robot operates using the Gymnasium [290] API. This environment features continuous observation and discrete action spaces, enabling realistic simulation of robotic decision-making.

The agent training and evaluation process is coordinated through scripts in the file `sb3_agent.py` that use Stable-Baselines3 and the Deep Q-Network [243] algorithm. These scripts automate training, evaluate the agents' performance across multiple episodes during training and validate the agents' predictions. Training hyperparameter configuration, evaluation settings and file paths are stored in a single file, `config.py`, ensuring consistency and ease of management.

To support deployment and scaling, the framework uses a Docker [33]-based workflow defined in the `Dockerfile` and coordinated by the `deploy.sh` deployment script. Template-based low-code CI/CD specifications, provided as input to LLMs, are available as `spec.txt` files and include various pipeline stages.

Our derived case study project emulates the required pipeline stages. The `test` stage lints the Python source files, then runs unit tests. The `train` stage trains a model and provides it as an artefact to subsequent stages. The `model_test`, `eval` and `release` stages run in parallel and utilise the model artefact provided by `train`. The `model_test` stage validates the trained model's predictions and output structure, whereas `eval` evaluates the model based on its mean reward and standard deviation. The `release` stage builds a Docker container image containing the model and an executable application that runs it. Finally, the `deploy` stage deploys the model.

Unit tests in `test_factory_robot_env.py`, `test_sb3_agent.py` and `test_sb3_agent_runtime.py`, help ensure reliability. These tests validate the environment's behaviour, agent logic and runtime execution. Linting helps maintain code quality by enforcing rules defined in the file `.pylintrc`, helping enforce best practices and standards.

11.4.2 Evaluation Scenarios

We defined four sequential evaluation scenarios corresponding to use cases in a typical development flow of a practitioner developing an RL CI/CD pipeline. In ***Create a New Pipeline***, an initial version of a new pipeline configuration is specified by the user and created by the LLM based on the specification.

For ***Edit the Pipeline***, the practitioner makes initial improvements to the specification. Artefacts are set to expire in one week, the model path environment variable is changed, and the `gpu-runner` tag is specified in all stages except `deploy`.

In the subsequent scenario, ***Add to the Pipeline***, the practitioner decides to improve the specification by making further changes. A `release` stage is added to the specification, and its corresponding source files are added to the repository. Unit tests and their Python sources are also added, and the user specifies that the `release` stage should not include a tag identifying the runner.

Finally, in the ***Remove from the Pipeline*** scenario, the practitioner decides that the `release` stage does not belong in the model pipeline, so they remove it. The practitioner also removes any `release`-relevant specification text and files from the repository and tidies up the specification, removing unused environment variables and an unnecessary configuration file.

11.4.3 Evaluation Variables

For the evaluation, we designed evaluation variables that were specific enough to be usable in the empirical evaluation of our approach, yet general enough to be applied to any MLOps or RLOps pipeline with minimal to no modification. We defined them in a structured manner based on our knowledge of RLOps pipeline requirements, combining domain-specific knowledge of RL processes and LLM applications, and decomposing relevant aspects of RLOps pipelines into quantifiable, discrete items that reflect relevant pipeline properties and behaviour. Siemens has verified that our evaluation variables accurately reflect the critical aspects of RLOps pipelines in an Industry 4.0 context. The result was 31 Boolean variables in three categories: precisely-defined generation tasks (Table 11.1), natural language tasks (Table 11.2) and runtime tasks (Table 11.3).

The precisely-defined generation tasks are based on technical requirements that represent objective, quantifiable standards, such as the correct selection of Python images and the correct order of pipeline stages, which are the core functional requirements of working CI/CD pipelines. On the other hand, natural language tasks involve fewer explicit imperatives and instead require interpretive comprehension of context-related natural language text, such as detecting dependency relations between jobs, and policies governing artefact management procedures. Lastly, the runtime task evaluation variables measure the success of pipeline execution, hence assessing whether the generated configurations work as desired in a real pipeline run.

Even though these evaluation variables have been designed to fit the RLOps pipeline generation space, they apply to MLOps in general, and the three categories provide a generalisable approach to benchmarking pipeline generation performance. The variables can be reused or extended by adapting domain-specific requirements without changing the pipeline generation architecture and evaluation approach.

Table 11.1: Description of Precisely-Defined Generation Tasks

Evaluation Variable	***Description***
GPI	The generated script uses a correct Python image.
GAS	The generated script contains all required stages.
GSO	The required stages are in the correct order.
GAR	All Python requirements and the necessary Linux packages are installed for all required stages correctly.
GEN	All specified environment variables are set correctly.
GEV	Unused environment variables were removed.
GMA	Correct creation of model artefacts.
GCD	Required Docker images are created correctly.
GDA	All Docker artefacts are created correctly.
GRA	A `release` stage was added.
GRU	Model runtime-relevant changes were added.
GRR	The `release` stage was removed.

11.4.4 LLMs

Our evaluation uses seven popular LLMs: **OpenAI GPT-3.5 Turbo** (Version 2023-06-13), **OpenAI GPT-4o** (Version 2024-08-06), **Qwen2.5 72B**, **Qwen2.5 Coder 32B**, **Llama 3.3 70B**, **Code Llama 70B** and **DeepSeek R1 70B** [286]. We ran the two OpenAI models on Microsoft Azure and ran the five freely available models in Ollama on a Cisco UCS C245 M6 server with an NVIDIA A100 80GB GPU, two AMD EPYC 7543 32-core processors and 512GB of RAM.

Using the five Ollama models is advantageous because they are free and require low resources. In contrast, the OpenAI models cost more but are thought to perform better according to benchmarks. We selected the LLMs from the standard benchmark EvalPlus [287], which is based on peer-reviewed studies and provides a leaderboard of LLMs for code generation tasks. We chose GPT-3.5 Turbo and an enhanced version (GPT-4o), and for the free models, we selected base models (Qwen2.5 72B and Llama 3.3 70B) and coding-specific related models (Qwen2.5 Coder 32B and Code Llama 70B). We also included DeepSeek R1 70B, as it is relatively new and popular. For the free models, we selected only those with over one million pulls on Ollama.

We adopt an LLM-agnostic approach to ensure flexibility in switching models based on cost, performance or future advancements in LLMs, without requiring fine-tuning. To enable a fair comparison, all models were set to a temperature of 0.7, low enough to ensure syntactically correct code whilst allowing varied solutions. Minimum response tokens were set to 10 to prevent short responses, and maximum response tokens were set to the capabilities of each model (GPT-3.5 Turbo: 4096, CodeLlama: 16000, GPT-4o: 16384, Qwen models and DeepSeek: 128000) to allow complex multi-stage pipeline configurations.

Table 11.2: Description of Natural Language Tasks

Evaluation Variable	***Description***
NTA	The `train` job creates artefacts in a specific directory.
NMT	The `model_test` job needs the `train` job and its artefacts.
NET	The `eval` job needs the `train` job and its artefacts.
NRT	The `release` job needs the `train` job and its artefacts.
NRD	The `deploy` job needs the `train` job and its artefacts.
NDA	The `deploy` job provides Docker results as an artefact.
NDI	Docker in Docker (DIND) is realised, and the correct image is selected.
NAR	Artefacts expire in one week.
NTD	No tag is specified for the `deploy` job.
NTR	No tag is specified for the `release` job.
NTO	`gpu-runner` is specified as the tag for all other jobs.
NBS	The `before_script` is correct for each job.

Table 11.3: Description of Runtime Tasks

Evaluation Variable	***Description***
RTL	Linting ran successfully.
RUT	Unit tests ran successfully.
RTR	Model training ran successfully.
RTE	Model tests ran successfully.
REV	Model evaluation ran successfully.
RRE	Release ran successfully.
RDP	Model deployment ran successfully.

11.4.5 Evaluation

We executed each scenario three times for each LLM and calculated the average error rate for each evaluation variable *var* as given by Equation 11.1:

$$\overline{E}_{\text{var}} = 1 - \frac{1}{n}\sum_{i=1}^{n} S^{i}_{\text{var}} \tag{11.1}$$

where n is the number of iterations ($n = 3$ in this case study) and S^{i}_{var} is the success rate (valued 0 or 1) for the variable *var* at iteration i.

We also calculated the average error rate for each evaluation variable category (precisely-defined generation tasks, natural language tasks and runtime tasks) using Equation 11.2:

$$\overline{E}_{\text{cat}} = 1 - \frac{1}{nm}\sum_{i=1}^{n}\sum_{j=1}^{m} S^{i,j} \tag{11.2}$$

where n is the number of iterations ($n = 3$), m is the number of variables in the category ($m = 12$ for precisely-defined and natural language tasks, and $m = 7$ for runtime tasks) and $S^{i,j}$ is the success rate (valued 0 or 1) on iteration i for variable j in the category.

As mentioned in Section 11.4.3, the evaluation variables are Booleans. Thus, their individual success rates for a given iteration are calculated by recording a value equivalent to false or true: 0 (false) if the output was incorrect or a runtime error occurred and 1 (true) if the output was correct and no error occurred.

The specific checks depend on the nature of the evaluation variable: values and entries in the generated pipeline configuration file are compared with our reference configurations, whereas for runtime tasks, the behaviour of the different pipeline steps at runtime is examined. For instance, for *GPI* (**the generated script uses a correct Python image**), we recorded 0 if no Python image was specified or an incorrect one was used and 1 otherwise. For *NTA* (**the `train` job creates artefacts in a specific directory**), we noted 0 if the train stage did not specify any artefacts or omitted a particular directory and 1 otherwise. As a final example, for *RTL* (**linting ran successfully**), we recorded 0 if linting did not execute or if linting failed and 1 otherwise. The averages were later subtracted from 1 to calculate the average error rates. The potential for bias was reduced via cross-checking between members of the author team. No ambiguous results were encountered due to the objective nature of the evaluation variables.

Note that this evaluation would not be performed as a regular part of the practitioner workflow (described in Section 11.3.5) – it is only necessary for evaluating our tool and the performance of different LLMs. It would only need to be performed again if the tool's implementation changed, variables were redefined, removed or added, or other LLMs were to be tested.

11.4.6 Results

Table 11.4 summarises the best-performing LLMs with average error rates of 0. GPT-4o scored 0 for each evaluation variable. Thus, it exhibited perfect performance across all evaluation variable categories and scenarios. Qwen2.5 Coder followed with an average error of 0.05 and GPT-3.5 Turbo with 0.07. Qwen2.5 Coder performed well at generation and the simpler natural language and runtime tasks, whereas GPT-3.5 Turbo excelled at the *Create* and *Edit* scenarios.

The worst-performing LLM overall was Code Llama, with an average error rate of 0.32, followed by DeepSeek (0.21) and Llama 3.3 (0.1). Qwen 2.5 achieved an average error rate of 0.09. Code Llama did not achieve an average error rate of 0 for any evaluation variable, so it does not feature in Table 11.4. It performed particularly poorly at runtime tasks. DeepSeek struggled with runtime and natural language tasks, whereas Llama 3.3 struggled with natural language tasks, though its average error rate of 0.1 remained acceptable. Our replication package [289] contains full results in the various `eval_results.xlsx` files in subdirectories of `_evaluation/`.

In summary, our approach utilised GPT-4o, Qwen2.5 Coder, GPT-3.5 Turbo, Qwen 2.5 and Llama 3.3 to successfully generate GitLab CI/CD pipeline configuration files with an average error rate of at most 0.1. Code Llama and DeepSeek, on the other hand, were not well-suited to our current pipeline generation approach. Note that the error rates we report in detail in our replication package [289] are likely an overestimation, as if a pipeline stage fails, all subsequent stages are also deemed to have failed. However, in

Table 11.4: The Best-Performing LLMs – Error Rates of 0 (Detailed Results in our Replication Package [289])

Evaluation Variable Category	***Create Pipeline***	***Edit Pipeline***	***Add to Pipeline***	***Remove from Pipeline***	***Average Error Rate***
Precisely-Defined Generation Tasks	GPT-3.5 Turbo GPT-4o Qwen2.5 Qwen2.5 Coder Llama 3.3	GPT-3.5 Turbo GPT-4o Qwen2.5 Qwen2.5 Coder Llama 3.3 DeepSeek	GPT-4o Qwen2.5 Coder Llama 3.3	GPT-3.5 Turbo GPT-4o Qwen2.5 Coder Llama 3.3 DeepSeek	GPT-4o Qwen2.5 Coder Llama 3.3
Natural Language Tasks	GPT-3.5 Turbo GPT-4o Qwen2.5	GPT-3.5 Turbo GPT-4o Qwen2.5 Qwen2.5 Coder	GPT-4o	GPT-3.5 Turbo GPT-4o	GPT-4o
Runtime Tasks	GPT-3.5 Turbo GPT-4o Qwen2.5 Qwen2.5 Coder	GPT-3.5 Turbo GPT-4o Qwen2.5 Qwen2.5 Coder DeepSeek	GPT-4o Llama 3.3	GPT-4o	GPT-4o
Average Error Rate	GPT-3.5 Turbo GPT-4o Qwen2.5	GPT-3.5 Turbo GPT-4o Qwen2.5 Qwen2.5 Coder	GPT-4o	GPT-4o	**GPT-4o**

practice, some of them may have succeeded had the first stage not failed. For instance, the unit tests may not have run successfully, but model deployment would have succeeded had the pipeline execution reached that stage. Furthermore, some evaluation variables are related, leading to multiple failures being recorded due to a single cause.

11.4.7 Evaluation Variables: Observations

Some recurring errors committed by the LLMs emerged during the evaluation. Below, we report evaluation variables or errors that occurred more than three times and affected the results of more than one scenario or LLM.

GAR (**all Python requirements and the necessary Linux packages are installed for all required stages correctly**): LLMs often ignored the instruction to install Python requirements using the `requirements.txt` file, instead devising their own set of requirements and version numbers, neglecting to install `pylint` or adding flags to commands (for example, adding `-user` to `pip install -r requirements.txt`). These mistakes led to additional errors, such as the `pylint` or `python` commands being unavailable, causing their respective stages to fail.

NBS (**the `before_script` is correct for each job**): The specifications dictate that the `test`, `train`, `model_test` and `eval` stages use a globally-scoped `before_script` that installs Python requirements. The `release` (used in the *Add to the Pipeline* scenario) and `deploy` stages should override the `before_script` with a blank `before_script` so as not to install any Python requirements since they are not needed for these stages. Whilst GPT-4o succeeded in this task without fail, many generation runs for other LLMs were unsuccessful. For instance, `release` and `deploy` often did not override the global `before_script` or preceding stages were erroneously configured to do so. Another widespread error was `pylint` being executed in the `before_script` section, even though the LLM was not instructed to do so.

RTR (**model training ran successfully**) and *RTE* (**model tests ran successfully**): The most common runtime error for variables was failure of model training and testing due to Python requirements not being installed. This mistake was always a secondary error caused by, for example, the incorrect Docker image being used (and consequently the `python` command not being available) or the LLM not installing prerequisites specified in the `requirements.txt` file.

11.5 Discussion

RQ$_{5.3}$: *How accurately can a low-code, template-based, modular flow approach using LLMs generate Industry 4.0-specific RLOps pipeline configurations?* Our approach was highly effective at generating RLOps pipelines in our Industry 4.0 RLOps case study, with low error rates. The high level of accuracy across five LLMs, including a perfect error rate of 0 for GPT-4o, is evidenced by our empirical evaluation.

Preliminary, informal experimentation before planning, designing and implementing our approach yielded inferior results. The strongest LLMs were unable to produce acceptable

pipeline configurations based on detailed textual instructions. The configurations produced in these initial attempts were consistently flawed, containing basic structural errors, incorrect values, invalid formatting and other issues that rendered them unsuitable for use without manual revision. These observations are consistent with results reported in other recent related work [267], [287].

To put the effectiveness of our approach into perspective, we informally compared it with the traditional manual approach to developing pipeline configurations. Based on feedback from Siemens, manually configuring similar RLOps pipelines typically takes many hours for both experienced DevOps engineers and domain experts. Our template-based method accurately generates pipeline configurations and validates them in around two to three minutes once the simple low-code template is filled out.

The patterns of error we have observed in our evaluation further support the effectiveness of our approach. The most common errors (occurring for *GAR*, *NBS*, *RTR* and *RTE*) are mostly related to dependency installation and environment setup. The same errors can occur during manual configuration and are hard to debug. Our approach's ability to establish flawless generation using GPT-4o implies that these common pitfalls can be eliminated with an appropriate LLM. The converse holds for the traditional approach since human error can occur regardless of expertise.

Using natural language specifications is highly beneficial in Industry 4.0, where domain experts possess comprehensive knowledge of production processes and RL requirements but may not have specific DevOps training. Traditional cloud-native MLOps solutions often assume technical skill sets that are not commonly found in industrial environments, posing an obstacle. The democratisation of RLOps deployment through our design approach enables production engineers and data scientists to express pipeline requirements without requiring fluency in CI/CD configuration syntax.

Our method addresses the special challenge of Industry 4.0 deployment in heterogeneous factory systems with different constraints. Applying Siemens' platform requires dozens of similar yet distinct RLOps pipelines to be used and operated across heterogeneous facilities – a fundamental scalability issue that our solution addresses. Unlike traditional approaches, which require multiple DevOps specialists to develop multiple pipeline variants, our tool can produce numerous variants using a single template specification, potentially decreasing resource requirements and implementation time by orders of magnitude.

Our approach enhances quality and consistency. Manual setup is highly variable and error-prone, leading to discrepancies across configurations. Our empirical findings indicate that successful LLMs produce highly consistent configurations, which is essential in an industrial context where operational efficiency requires standardised, reliable automation.

The effectiveness of our approach extends beyond the case study context. The evaluation scenarios we defined are generalisable to other MLOps and RLOps scenarios, and the evaluation variable categories (precisely-defined generation tasks, natural language tasks and runtime tasks) provide a framework for evaluating effectiveness across different pipeline generation scenarios. The robust performance across all three categories suggests that our approach can address both the technical precision required for functional pipelines and the interpretive complexity of real-world deployment scenarios.

RQ$_{5.4}$: *How do popular LLMs perform in generating Industry 4.0-specific RLOps pipeline configurations, and do they commit or encounter any recurring errors or challenges?* Our analysis demonstrates that OpenAI's GPT-4o model performed best across all task types, achieving an error rate of 0 on every evaluation variable. Qwen2.5 Coder and GPT-3.5 Turbo also performed exceptionally well. Qwen 2.5 and Llama 3.3 are also suitable for our case study, with low variability in their error rates and high consistency, reliability and accuracy in generating correct outputs. DeepSeek and CodeLlama did not perform very well. Considering that all seven LLMs used identical pipeline specifications and five performed well, we do not suspect that our approach is the reason two LLMs performed poorly. We assume it would take inordinate effort to get the two poorest-performing LLMs to work with an average error rate of at most 0.1 to match the worst-performing of the other five. Without a compelling reason for using DeepSeek or CodeLlama specifically, it would unlikely be worth the effort.

Notably, standard (non-coder) LLMs often outperformed specialised code generation LLMs. One of the main takeaways from our results is that one should not assume that code generation LLMs will perform better than general LLMs for code generation tasks. In this regard, our results stand in stark contrast to results reported in other work [267], [287]. This discrepancy may have several potential sources. It may result from our unique evaluation approach, as code generation LLMs may be overfitting to standardised benchmark tests, and popular evaluation datasets like HumanEval can substantially overlap with the training corpora. Another explanation may be due to differences in the nature and complexity of the evaluated tasks. Alternatively, our template-based natural language approach may produce differing results due to our retry logic.

Few recurring mistakes or errors were encountered. The main errors occurred during the installation of Python requirements, setting the `before_script` for some stages and running the model training and model testing stages. The latter two errors were usually caused by the former one. GPT-4o worked perfectly, so there was no strong argument for fixing errors to improve our results marginally.

The success of our evaluation has ramifications for scaling RLOps. The low error rates achieved by multiple LLMs on different platforms and from different vendors mean organisations are not tied to a single LLM provider, providing flexibility for cost optimisation and risk mitigation. The template-based specification enables the capture and reuse of knowledge across projects, allowing organisations to build libraries of validated pipeline patterns for new deployments.

11.6 Threats to Validity

In this section, we discuss threats to validity according to Wohlin et al. [85].

Construct Validity The effectiveness of natural language-based specification components depends heavily on users' correct specifications. The validity of the results could be affected when human inputs vary, leading to inconsistent outputs. We conducted multiple runs with different LLMs and scenarios to smooth out unexpected outcomes.

External Validity The single RL industry case we examine may not be transferable

to other industrial projects. Our limited access to RL projects in the industry could lead us to miss relevant pipeline engineering and RLOps practices. However, our analysis of the relevant practitioner literature and discussions with Siemens suggest that the use of RL in this study is typical, which mitigates this threat to external validity.

Internal Validity Our results depend on the completeness of the specification template, the correctness of completed specification template instances and the LLMs' interpretations of the provided specifications. Our template demonstrates both sufficient generality and specificity to produce correct results across multiple LLMs, as evidenced by the evaluation.

Conclusion Validity The results of our experiments rely on a single case study and a limited number of use cases. Broadening the study's scope by including more industry projects and use cases could strengthen the validation of our approach and conclusions. Since we have limited access to industry projects, we addressed what we could by creating use cases similar to CRUD (create, read, update and delete) operations and conducting evaluations with multiple LLMs.

11.7 Conclusion and Future Work

Our study presents an approach to automatic pipeline configuration in Industry 4.0 that breaks new ground in the field. Our work includes contributions that combine correct pipeline configuration generation from our template-based implementation with rigorous empirical evaluation of our tool and popular LLM capabilities to transform template-based specifications into RLOps CI/CD pipeline instances.

Our tool reduces the need for specialist DevOps knowledge by providing templates that guide users, eliminating the need to learn how to write and debug GitLab pipeline configurations directly. It can also potentially address bottlenecks in RLOps CI/CD pipeline development, reducing development time.

We identified common errors that different LLMs make when generating RLOps pipelines, which can guide practitioners in improving communication with LLMs when implementing an approach like ours. A key takeaway is that the limitations of the poorly performing LLMs do not appear to be due to ambiguity or lack of detail in the pipeline specifications, since most LLMs performed satisfactorily.

Our work fills an urgent gap in the literature, presenting the first empirical evidence of the systematic generation of RLOps pipelines for Industry 4.0. Its implications extend to the broader AI engineering community. Our empirical findings offer methodological insights for AI engineering research, particularly regarding the selection of LLMs for infrastructure code generation tasks. The surprising result is that general-purpose LLMs often outperform code-specialised models for this domain. The transferability of our evaluation framework provides a reusable methodology for assessing the quality of automated code generation in other DevOps contexts, including infrastructure-as-code, containerisation and cloud deployment automation. This transferability results in a practical research tool for future studies in AI-assisted software engineering.

Future work would involve further integrating our approach into Siemens' system, adding improved model life cycle management patterns and practices to generated pipelines,

and assessing the effectiveness when encountering new pipeline and project features. We also plan to test our approach in a user study, comparing it with manual creation of CI/CD pipeline configurations.

Part VII

Conclusion

12 Conclusion

12.1 Research Questions Revisited

In Sections 1.3 and 1.4 we outlined research problems **RP$_1$**-**RP$_5$** and corresponding research questions **RQ$_1$**-**RQ$_5$** respectively. Below, we revisit each research question and summarise how it was addressed by the research contributions **RC$_1$**-**RC$_{10}$** described in Section 1.5 and publications **P$_1$**-**P$_{10}$** respectively.

Research Question 1 (RQ$_1$)

What ADDs, options, decision drivers and relations are relevant for the ML workflow and deployment when productionising ML models?

RQ$_{1.1}$ *Which ADDs are available to choose from in the context of the ML workflow, and what are their corresponding decision options?*

RQ$_{1.2}$ *What are the relations between these decisions and their decision options?*

RQ$_{1.3}$ *Which decision drivers (forces) are relevant to the decision options?*

RQ$_{1.4}$ *Which ADDs are available to choose from in the context of deployment for ML, and what are their corresponding decision options?*

RQ$_{1.5}$ *What are the relations between these decisions and their decision options?*

RQ$_{1.6}$ *Which decision drivers (forces) are relevant to the decision options?*

In Chapter 2, we investigated which ADDs shape the ML workflow by analysing 29 practitioner-oriented grey-literature sources with Straussian Grounded Theory. We coded the material into an ADD model containing 10 decisions, 43 decision options and 30 decision drivers, covering areas such as data processing automation, feature storage, batched vs streamed data processing, ingestion modes, pipeline triggering, model-building pipelines and AutoML. These decisions and forces were formalised in a Python-based metamodel and expressed as UML diagrams, giving a structured view of the design space and the relations between decisions and options. We used theoretical saturation and traceability from options and drivers back to the 29 sources as the main evidence that the model captured practitioners' current practices and thus addressed the research questions.

Chapter 3 examined ADDs specific to deploying ML systems by applying Straussian Grounded Theory to 35 grey literature sources. From this corpus, we derived an ADD model with seven deployment decisions, 26 decision options and 44 decision drivers,

formalised in Python with generated UML diagrams and tabular summaries. The decisions include how far to automate deployment (from no automation through scripts to CI/CD), how to structure integration and delivery, what tasks a delivery pipeline should perform, how to trigger pipelines, how to version data, how many model versions to run and whether and when to adopt MLOps. By mapping each option and decision driver back to concrete evidence in the source set and discussing usage scenarios and threats to validity, we showed how this model supported reasoning about deployment trade-offs.

Research Question 2 (RQ_2)

To what extent do MLOps practices and ADDs apply to training strategies and architectures in RL systems, particularly within CPPSs?

$RQ_{2.1}$ *Which patterns and practices do practitioners currently use for training strategies in RL architectures?*

$RQ_{2.2}$ *What are the relations between existing training patterns and practices? Specifically, which ADDs are pertinent when designing RL systems?*

$RQ_{2.3}$ *What are the influencing factors (i.e. decision drivers) in architecting RL systems in the eyes of the practitioner today?*

$RQ_{2.4}$ *To what extent are known ML and RL ADDs applicable to RL in a CPPS context?*

$RQ_{2.5}$ *Which ADD decision options typically used in ML and RL apply directly to a real-world CPPS that utilises RL?*

$RQ_{2.6}$ *Is it necessary to adapt ADD decision options typically used in ML for use with RL in a CPPS context?*

In Chapter 4, we focused on ADDs for training strategies in RL systems, analysing 33 practitioner sources with a model-based Straussian Grounded Theory method. The resulting ADD model encapsulated six top-level decisions, 29 decision options and 19 decision drivers, covering issues such as model architecture, RL model training, RL checkpoints, transfer learning in RL, RL distribution strategies and RL hyperparameter tuning. These elements were coded in Python, with explicit relations between decisions and options. By tracking when additional sources no longer introduced new decision or relation types and by checking coverage across all 33 sources, we argued that this model adequately reflected current RL training practice and thus directly addressed questions about prevalent patterns, their relationships and their influencing forces.

Our case study of Chapter 5 investigated how ML/RL decision models from the preceding three chapters transferred into an RL-based CPPS by reusing our existing catalogue of 22 decisions and 86 decision options. We examined 15 project artefacts from a single Industry 4.0 RL system – source code, CI/CD pipelines, informal documentation and UML diagrams – and modelled the system using a Python metamodel and UML component and pipeline views, then systematically mapped each of the 22 decisions and

86 options onto concrete elements of the case. Each decision and option was classified as used as-is, used in adapted form for RL, not present, or not applicable, and these classifications were cross-checked amongst the authors and validated with the industrial partner. The resulting counts per category made it clear which parts of the MLOps/RL catalogue carried over unchanged, which required RL-specific adaptations, which did not occur and which did not apply in this RLOps setting, thereby answering the research questions on transferability and gaps.

Research Question 3 (RQ_3)

To what extent do semi-formal MLOps, DL and RL architecture models and diagrams enhance the understandability of MLOps, DL and RL-based systems for developers?

$RQ_{3.1}$ *How does the exclusion or inclusion of semi-formal MLOps system diagrams affect the accuracy with which participants assess their performance in the context of their perceived task correctness?*

$RQ_{3.2}$ *To what extent does the provision of semi-formal architecture models, in addition to system source code, improve the understandability of DL and RL patterns and practices?*

In Chapter 6, we investigated whether semi-formal UML-based models improved the understandability of MLOps architectures by modelling three real-world MLOps systems from major cloud providers and conducting a between-subject controlled experiment with 63 participants. For each system, we produced component and pipeline diagrams based on our metamodel. We paired them with the providers' original informal descriptions, then randomly assigned participants to either a control group that saw only the informal material or an experimental group that saw both the informal and UML views. Participants completed architecture-understanding tasks for the three systems, whilst we measured task correctness and duration as dependent variables. Using robust statistical tests, we found statistically significant improvements in correctness for the UML-diagram group, no significant increase in task duration and a significant correlation between time and correctness only when the semi-formal UML diagrams were present, thereby addressing the hypotheses about effectiveness and efficiency.

Chapter 7 described a study examining how semi-formal models affected understanding of DL and RL practices by running a within-subjects experiment with 158 software engineering students across two systems (one DL and one RL). Each participant solved tasks under two conditions (source-code-only versus source-code-plus-UML-models). We recorded correctness, completion time and self-reported confidence for each. For the DL tasks, the control condition achieved an average correctness of 0.7121, compared with 0.7759 when models were provided, whilst average task durations were 1571.62 seconds versus 1763.85 seconds, a difference that was not statistically significant; for the RL tasks, the averages were 0.6808 versus 0.6612 in correctness and 1883.80 versus 1925.46 seconds in duration, again without significant time differences. These quantitative results, analysed with appropriate non-parametric tests, showed that models significantly

improved understanding for the DL system but not for the RL system, clarifying when such representations help.

Research Question 4 (RQ_4)

How can conformance to MLOps and RL ADDs and best practices - particularly automation support and RL training methods - be objectively assessed and automated?

$RQ_{4.1}$ *How can MLOps systems' support for the core quality aspect of automation be assessed in the context of ADDs and their respective decision options?*

$RQ_{4.2}$ *What types of metrics can be applied to evaluate these levels of support, and how effective are they?*

$RQ_{4.3}$ *How effectively can LLMs assess best practices in RL training pipelines compared to heuristic-based code detectors?*

$RQ_{4.4}$ *What are the key architectural rules required for compliance with the best practices for checkpoints, hyperparameter tuning, training/agent configuration and single vs multi-agent RL in RL-based systems?*

Chapter 8 proposed a model-driven and metrics-based approach to assessing automation as a quality aspect in MLOps architectures by combining earlier decision models with a new empirical dataset of 22 system architectures. We selected three automation-related ADDs ("How to Automate Integration and Delivery in an ML Context?", "How to Trigger a ML Pipeline or Orchestrator?" and "How to Automatically Process the Data Used for Model Building?") and modelled 22 vendor MLOps systems and system variants against a shared metamodel, then manually rated each system on an ordinal scale to create a ground truth. On this basis, we defined and implemented detectors for new, technology-independent metrics over the models and built three ordinal regression models – one per ADD – to predict the manual ratings from metric combinations. The regression results showed high predictive accuracy for all three ADDs, demonstrating that a relatively small set of basic metrics over 22 systems was sufficient to approximate manual judgements of automation support and thereby answer the research questions.

In Chapter 9, we explored rule-based assessment of RL training practices by defining 31 architectural rules and evaluating detectors across one industrial RL-based CPPS and 10 open-source RL projects. We derived the 31 rules – for checkpoints, hyperparameter tuning, training configuration and agent configuration – then implemented two detector types: heuristic pattern-matchers and LLM-based detectors, with each rule corresponding to a detector implementation. Applying each detector type across the 11 case studies, we measured precision, recall and F_1-scores per rule group and case study; for the industrial system, for example, LLM-based detectors achieved F_1-scores of 0.933 for checkpoint rules, 0.909 or hyperparameter-tuning rules and 0.909 for training/agent-configuration rules, compared with 0.545, 0.8 and 0.667, respectively, for heuristics. These quantitative comparisons showed that LLM-based assessment consistently achieved higher recall and F_1 scores across many settings, especially for complex code patterns, thereby addressing the

research questions about effectiveness relative to heuristics and the concrete rules required.

Research Question 5 (RQ_5)

How accurately can LLM-powered low-code approaches generate and validate correct MLOps/RLOps pipeline configurations for RL workflows, particularly in Industry 4.0 contexts?

$RQ_{5.1}$ *How accurately can a low-code, template-based modular flow approach – leveraging LLMs for generation and validation – produce correct MLOps pipeline configurations for RL?*

$RQ_{5.2}$ *How accurately do different LLMs generate MLOps pipeline configurations for RL, and what are their limitations?*

$RQ_{5.3}$ *How accurately can a low-code, template-based, modular flow approach using LLMs generate Industry 4.0-specific RLOps pipeline configurations?*

$RQ_{5.4}$ *How do popular LLMs perform in generating Industry 4.0-specific RLOps pipeline configurations, and do they commit or encounter any recurring errors or challenges?*

Chapter 10 examined how low-code techniques and LLMs could generate MLOps pipelines for RL systems by combining template-based design with an empirical evaluation over seven LLMs and three open-source RL projects. We decomposed the RL MLOps workflows of the three projects into modular filters and a coordinating pipeline, defined pipeline specifications, and asked seven different LLMs to generate configurations from these templates. We then calculated an average configuration error rate of 0.187 across all models. In the initial run, OpenAI GPT-4o produced the lowest error rate at 0.09, followed by Qwen2.5 Coder at 0.15, whilst other LLMs exhibited higher rates. We then refined the templates and implementation based on these observed errors. Re-evaluating the improved setup with GPT-4o alone yielded an error rate of 0 in the reported experiments, showing that with suitable templates and a strong LLM, the low-code approach could generate perfectly correct pipelines for the three RL projects and thus meet the research goals.

Finally, Chapter 11 extended the low-code and LLM-assisted pipeline generation approach to RLOps in Industry 4.0 by modelling RLOps workflows with a Pipes-and-Filters architecture and empirically comparing seven LLMs across four representative RLOps use cases. We defined template-based specifications for typical RLOps concerns – training, testing, deployment and monitoring – and used the seven LLMs to produce concrete pipeline configurations for the four use cases derived from a smart factory CPPS. The generated pipelines were executed end-to-end, and we recorded configuration and runtime errors per LLM and use case, allowing us to compare error profiles and to identify which models consistently produced correct, executable pipelines. Our method achieved perfect results, showing that with suitable LLM choices, error rates were low enough to support the reliable automated creation of RLOps pipelines, reducing manual DevOps effort and addressing the research questions about feasibility and effectiveness in an

Industry 4.0 context.

12.2 Overarching Approach Revisited

In Section 1.9, we described our overarching approach. We noted that some of the solutions we developed in early studies, as part of our overall approach to address the research problems described in Section 1.3, could be enhanced by recent advances in AI. This section discusses how AI could potentially enhance certain manual aspects of our overarching approach.

ML models can now scan technical documentation, find architectural patterns and decisions made in the design, and group system elements based on source code and architectural diagrams[1][2]. Recent studies show that LLMs and generative AI can significantly improve Grounded Theory analysis by automating coding, finding patterns, and extracting themes without compromising the accuracy of qualitative work. For example, Übellacker [291] describes AcademiaOS, an initial attempt to automate Grounded Theory using LLMs by applying LLMs' understanding, generation and reasoning capabilities to augment humans in qualitative research. Yue et al. [292] apply ChatGPT-4 Turbo to Grounded Theory and describe how LLMs can enhance the efficiency of text coding and qualitative analysis in Grounded Theory. When gathering **Architectural Knowledge** (**RC$_1$**-**RC$_4$**), rather than periodically performing manual Grounded Theory and content analysis studies, which can be time-consuming, we could reduce manual work by using AI tools to implement alternative solutions. An advantage is that this knowledge-gathering step could be more easily automated within a CI/CD pipeline. The gathered knowledge could be automatically updated as necessary and used in later CI/CD steps that implement our approach and utilise it, such as in **RC$_5$**-**RC$_6$** (**Architectural Modelling**), **RC$_7$**-**RC$_8$** (**Architectural Conformance**) and **RC$_9$**-**RC$_{10}$** (**Pipeline Generation**).

LLMs can now build structural and behavioural models for systems without relying on human-written code. Latest findings also suggest that LLMs now enable systems to be modelled automatically, for example, with SysML diagrams, thereby reducing the need for manual work. Apvrille and Sultan [293] describe how LLMs can make sense of natural-language descriptions and generate content that fits predefined formats, thereby supporting the automation of model generation. They introduce a framework that enables LLMs to automatically construct structural and behavioural SysML diagrams from system specifications. **Architectural Modelling** (**RC$_5$**-**RC$_6$**) is currently a manual process in which the practitioner models the system architecture in Python using our extensible MLOps/RLOps metamodel. This formal system model can then be processed in later stages of the pipeline, for instance, during the **Architectural Conformance** (**RC$_7$**-**RC$_8$**) checks. Rather than generating models directly, we envision an enhanced solution that utilises generative AI to write Python code and update our Codeable Models [78] metamodel and formal system architecture models. This automated stage could dramatically reduce the initial required modelling effort in the early stages of a project

[1] https://www.westat.com/machine-learning-nlp

[2] https://mindthegraph.com/blog/automated-content-analysis

and subsequent updates to the model. Optionally, the tool could subsequently generate UML visualisations from the Python models using PlantUML [79] as it currently does.

RC$_7$ (**Architectural Conformance for MLOps Systems**) substantially reduces the need for manual assessment of MLOps/RLOps architectures for conformance to known ADDs and decision options and allows for automation in a CI/CD pipeline. However, some manual effort is still required to implement the detectors that analyse the models and the metrics that evaluate the support for quality aspects based on the applied decision options. An upgraded solution could use LLMs similarly to our implementation of **RC$_8$** (**Architectural Practices Conformance Check for RL**) – which is already integrated into Siemens' RLOps pipeline as-is – by examining the Python models generated using techniques from **RC$_5$**-**RC$_6$** (**Architectural Modelling**) based on the Architectural Knowledge from **RC$_1$**-**RC$_4$** (**Consolidation of Architectural Knowledge**). Indeed, recent related work describes the application of LLMs to architectural conformance checking. Keshri et al. [294] present a novel system for verifying code implementations against the algorithms and methodologies from corresponding research papers. Lewis et al. [295] use retrieval-augmented generation to extract relevant details from research papers and code repositories and compare them using LLMs to reduce manual effort, enhance research credibility and advance the state of the art in code verification.

12.3 Open Research Challenges

In collaboration with Siemens, we have identified several research challenges that require attention and corresponding architectural solutions. Owing to the combination of ML, RL and Industry 4.0 technologies, the challenges posed in software architecture require new solutions. This section examines four significant challenges in building reliable, scalable architectures for intelligent manufacturing systems and how AI may help address them.

Model Versioning for RL in CPPSs

Challenge Traditional methods for managing model versions are not suited to RL in smart factories due to agents' continuous learning, which calls for more advanced tools to oversee model updates, track policy changes and update models to fit the smart factory environment whilst maintaining operational stability. Difficulties arise when several agents are distributed across different parts of a production line and require simultaneous, jointly managed version updates and rollbacks.

Strategy In our prior work, we noted the use of a hybrid MLOps-RLOps solution that updates model registries with RL policy metadata, exploration strategies and environment interaction history [9]. Solutions involve using distributed version control to track model parameters and learned policies, as well as implementing rollback mechanisms [296]. AI-powered automated version management might choose the most effective model using AI meta-learning techniques, analysing prior performance records and the current context.

Communication Protocols for RL in Industry 4.0

Challenge Current communication schemas, such as the Model Context Protocol (MCP) [3], whilst useful for AI-tool interactions, do not cover the necessary architectural designs used in industrial fields, where critical operations, real-time constraints and legacy systems dominate. Protocol implementations primarily handle desktop applications and simple AI assistants. They do not meet the industry's needs, which require reliability, rigid response times and compatibility with current manufacturing systems.

Strategy One strategy is to work on system architectures built for industry, adding real-time guarantees, secure authentication and manufacturing-oriented MCP-like protocol architectures[4]. A solution could build federated networks that cross multiple security domains in a factory, use special tools for data transfer over industrial networks and facilitate interoperability of networks with other popular industrial protocols. AI could also help facilitate protocol optimisation as networks and production environments evolve.

Agentic AI Architecture for CPPSs

Challenge Using agentic AI in CPPSs means building architectures that can handle complex decisions on their own, whilst making sure the AI is secure, reliable and works with previous control systems [297][5]. Currently, agentic systems mainly concentrate on software environments and fail to include industry-relevant error handling, latency assurances and reliability guarantees[6]. It is essential to ensure that the changing behaviours of autonomous agents do not disrupt the orderly operations of manufacturing [148].

Strategy A strategy could emphasise architectures that ensure safety-related functions are under separate control from autonomous agents, by using advanced verification techniques for agentic systems and by developing hybrid human-agent supervision models [297]. Some ways to reduce risks are to precisely define agent and human responsibilities, standardise interfaces between agent systems and industrial controls, and to include continuous monitoring and intervention methods [296]. AI-assisted architectural optimisation could influence agent autonomy, given the risks and outcomes in production.

Support for RL in CI/CD Pipelines

Challenge Existing frameworks for CI/CD fail to adequately support RL systems since they do not support many of the unique needs of RL, like those described in the challenges above (versioning for RL models, support for RL communication protocols and agentic AI systems). The need is for CI/CD pipelines that understand complex deployment environments and can learn from system evolution [296], as traditional software deployment methods do not consider the unique demands of learning and adaptation after release [9].

Strategy One strategy is to build multi-step deployment systems for RL artefacts, create automated systems to test agent capabilities and design monitoring tools that notice drift

[3] https://www.anthropic.com/news/model-context-protocol

[4] https://www.linkedin.com/pulse/securing-model-context-protocol-mcp-architecture-best-srivastava-v4hgf

[5] https://techcommunity.microsoft.com/blog/machinelearningblog/baseline-agentic-ai-systems-architecture/4207137

[6] https://www.ibm.com/think/topics/agentic-architecture

in production environments [296]. AI-supported pipeline configuration could bootstrap the appropriate deployment methods based on the model type, risks and production needs, with feedback loops that enhance the system in controlled production settings [9].

12.4 Summary

This doctoral thesis describes our systematic approach towards supporting practitioners in developing ML and RL-based systems, particularly for Industry 4.0 CPPSs. We outlined five research problems and how we addressed them through Straussian Grounded Theory-based grey literature studies, the development of a novel ADD metamodel in Python, an exploratory industry case study on an Industry 4.0 CPPS, the development of a novel, formal metamodel for modelling ML and RL-based systems, controlled experiments testing whether providing semi-formal system and architecture representations as UML diagrams helps understanding of MLOps system architectures compared to informal system architecture diagrams, methods for automatically checking conformance of MLOps architectures to known ADDs, the assessment of conformance to RL best practices in RL code using LLMs and the generation of MLOps and RLOps pipeline architecture using low-code and LLMs. We showed where we have already applied AI techniques and described how we could further enhance our approach with AI.

Using our approach in CI/CD pipelines helps automate much of the process. Still, some significant open challenges remain: advanced methods for versioning models for distributed RL agents, effective communication protocols for industrial applications, the safe integration of agents into AI systems, and of RL into CI/CD pipelines. We described these four open challenges and possible strategies to address them. Future work should address the identified open challenges and further automate our approach in collaboration with industry partners. Our research provides a foundation for intelligent solutions that can continually adapt to meet the dynamic demands of modern production systems, ultimately facilitating the implementation of functional AI in industry when architecting and engineering ML and RL-based systems.

Bibliography

[1] V. Lakshmanan, S. Robinson and M. Munn, *Machine Learning Design Patterns*. O'Reilly Media, Inc., Oct. 2020, ISBN: 9781098115784.

[2] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[3] A. S. Bhatia, M. K. Saggi, A. Sundas and J. Ashta, "Reinforcement Learning", *Machine Learning and Big Data: Concepts, Algorithms, Tools and Applications*, pp. 281–303, 2020.

[4] D. Kreuzberger, N. Kühl and S. Hirschl, "Machine Learning Operations (MLOps): Overview, Definition, and Architecture", *IEEE Access*, vol. 11, pp. 31 866–31 879, 2023. DOI: 10.1109/ACCESS.2023.3262138.

[5] N. T. Hewage and D. A. Meedeniya, "Machine Learning Operations: A Survey on MLOps Tool Support", *ArXiv*, vol. abs/2202.10169, 2022. [Online]. Available: https://api.semanticscholar.org/CorpusID:247011728.

[6] M. Treveil et al., *Introducing MLOps: How to Scale Machine Learning in the Enterprise*. O'Reilly Media, Incorporated, 2020, ISBN: 9781492083290.

[7] P. Li et al., "RLOps: Development life-cycle of reinforcement learning aided open RAN", *IEEE Access*, vol. 10, pp. 113 808–113 826, 2022.

[8] E. Ntentos, S. J. Warnett and U. Zdun, "Supporting Architectural Decision Making on Training Strategies in Reinforcement Learning Architectures", in *2024 IEEE 21st International Conference on Software Architecture (ICSA)*, 2024, pp. 90–100. DOI: 10.1109/ICSA59870.2024.00017.

[9] S. J. Warnett and U. Zdun, "Bridging the Gap Between MLOps and RLOps: An Industry 4.0 Case Study on Architectural Design Decisions in Practice", in *2025 IEEE 22nd International Conference on Software Architecture (ICSA)*, 2025, pp. 232–242. DOI: 10.1109/ICSA65012.2025.00031.

[10] H. Singh and B. Singh, "Industry 4.0 technologies integration with lean production tools: a review", *The TQM Journal*, Feb. 2024. DOI: 10.1108/TQM-02-2022-0065.

[11] K. Höse, A. Amaral, U. Götze and P. Peças, "Manufacturing Flexibility through Industry 4.0 Technological Concepts—Impact and Assessment", *Global Journal of Flexible Systems Management*, vol. 24, pp. 271–289, 2023. [Online]. Available: https://api.semanticscholar.org/CorpusID:258150285.

[12] H. Lasi, P. Fettke, H.-G. Kemper, T. Feld and M. Hoffmann, "Industry 4.0", *Business & Information Systems Engineering*, vol. 6, no. 4, pp. 239–242, 2014, ISSN: 1867-0202. DOI: 10.1007/s12599-014-0334-4. [Online]. Available: https://doi.org/10.1007/s12599-014-0334-4.

[13] L. Monostori et al., "Cyber-physical systems in manufacturing", *CIRP Annals*, vol. 65, no. 2, pp. 621–641, 2016, ISSN: 0007-8506. DOI: https://doi.org/10.1016/j.cirp.2016.06.005. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0007850616301974.

[14] S. J. Oks et al., "Cyber-Physical Systems in the Context of Industry 4.0: A Review, Categorization and Outlook", *Information Systems Frontiers*, 2022. [Online]. Available: https://api.semanticscholar.org/CorpusID:248036052.

[15] L. Monostori, "Cyber-physical Production Systems: Roots, Expectations and R&D Challenges", *Procedia CIRP*, vol. 17, pp. 9–13, 2014, Variety Management in Manufacturing.

[16] A. Khdoudi, T. Masrour, I. El Hassani and C. El Mazgualdi, "A Deep-Reinforcement-Learning-Based Digital Twin for Manufacturing Process Optimization", *Systems*, vol. 12, no. 2, 2024, ISSN: 2079-8954. DOI: 10.3390/systems12020038. [Online]. Available: https://www.mdpi.com/2079-8954/12/2/38.

[17] A. Jansen and J. Bosch, "Software Architecture as a Set of Architectural Design Decisions", in *5th Working IEEE/IFIP Conference on Software Architecture (WICSA'05)*, IEEE, 2005.

[18] U. Zdun, R. Capilla, H. Tran and O. Zimmermann, "Sustainable Architectural Design Decisions", *IEEE Software*, vol. 30, no. 6, pp. 46–53, 2013. DOI: 10.1109/MS.2013.97.

[19] R. Kazman, J. Asundi and M. Klein, "Quantifying the Costs and Benefits of Architectural Decisions", in *Software Engineering, International Conference on*, Los Alamitos, CA, USA: IEEE Computer Society, May 2001, p. 0297. DOI: 10.1109/ICSE.2001.919103. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ICSE.2001.919103.

[20] O. Zimmermann, U. Zdun, T. Gschwind and F. Leymann, "Combining Pattern Languages and Reusable Architectural Decision Models into a Comprehensive and Comprehensible Design Method", in *Software Architecture, 2008. WICSA 2008. Seventh Working IEEE/IFIP Conference on*, 2008, pp. 157–166. DOI: 10.1109/WICSA.2008.19.

[21] L. Bass, I. Weber and L. Zhu, *DevOps: A Software Architect's Perspective*, 1st. Addison-Wesley Professional, 2015, ISBN: 0134049845.

[22] J. Humble and D. Farley, *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*, 1st. Addison-Wesley Professional, 2010, ISBN: 0321601912.

[23] S. J. Warnett and U. Zdun, "Architectural Design Decisions for the Machine Learning Workflow", *Computer*, vol. 55, no. 3, pp. 40–51, 2022. DOI: https://doi.org/10.1109/MC.2021.3134800.

[24] S. J. Warnett and U. Zdun, "Architectural Design Decisions for Machine Learning Deployment", in *2022 IEEE 19th International Conference on Software Architecture (ICSA)*, IEEE, 2022, pp. 90–100. DOI: https://doi.org/10.1109/ICSA53651.2022.00017.

[25] M. Testi et al., "MLOps: A Taxonomy and a Methodology", *IEEE Access*, vol. 10, pp. 63 606–63 618, 2022. DOI: 10.1109/ACCESS.2022.3181730.

[26] G. Symeonidis, E. Nerantzis, A. Kazakis and G. A. Papakostas, "MLOps - Definitions, Tools and Challenges", in *2022 IEEE 12th Annual Computing and Communication Workshop and Conference (CCWC)*, 2022, pp. 0453–0460. DOI: 10.1109/CCWC54503.2022.9720902.

[27] P. Ruf, M. Madan, C. Reich and D. Ould-Abdeslam, "Demystifying MLOps and Presenting a Recipe for the Selection of Open-Source Tools", *Applied Sciences*, vol. 11, no. 19, 2021, ISSN: 2076-3417. DOI: 10.3390/app11198861. [Online]. Available: https://www.mdpi.com/2076-3417/11/19/8861.

[28] T. Mboweni, T. Masombuka and C. Dongmo, "A Systematic Review of Machine Learning DevOps", in *2022 International Conference on Electrical, Computer and Energy Technologies (ICECET)*, 2022, pp. 1–6. DOI: 10.1109/ICECET55527.2022.9872968.

[29] Y. Luo, M. Raatikainen and J. K. Nurminen, "Autonomously Adaptive Machine Learning Systems: Experimentation-Driven Open-Source Pipeline", in *2023 49th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, Sep. 2023, pp. 44–52. DOI: 10.1109/SEAA60479.2023.00016.

[30] Y. Haviv and N. Gift, *Implementing MLOps in the enterprise*, en. Sebastopol, CA: O'Reilly Media, Dec. 2023.

[31] Y. Zhou, Y. Yu and B. Ding, "Towards MLOps: A case study of ML pipeline platform", in *2020 International Conference on Artificial Intelligence and Computer Engineering (ICAICE)*, Oct. 2020, pp. 494–500. DOI: 10.1109/ICAICE51518.2020.00102.

[32] S. Garg, P. Pundir, G. Rathee, P. Gupta, S. Garg and S. Ahlawat, "On Continuous Integration / Continuous Delivery for Automated Deployment of Machine Learning Models using MLOps", in *2021 IEEE Fourth International Conference on Artificial Intelligence and Knowledge Engineering (AIKE)*, 2021, pp. 25–28. DOI: 10.1109/AIKE52691.2021.00010.

[33] D. Merkel, "Docker: lightweight Linux containers for consistent development and deployment", *Linux journal*, vol. 2014, no. 239, p. 2, 2014.

[34] A. Vaswani et al., "Attention is All You Need", in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, ser. NIPS'17, Long Beach, California, USA: Curran Associates Inc., 2017, pp. 6000–6010, ISBN: 9781510860964.

[35] F. Khomh, B. Adams, J. Cheng, M. Fokaefs and G. Antoniol, "Software Engineering for Machine-Learning Applications: The Road Ahead", *IEEE Software*, vol. 35, no. 5, pp. 81–84, Sep. 2018. DOI: 10.1109/MS.2018.3571224.

[36] M. Kim, T. Zimmermann, R. DeLine and A. Begel, "Data Scientists in Software Teams: State of the Art and Challenges", *IEEE Transactions on Software Engineering*, vol. 44, no. 11, pp. 1024–1038, 2018. DOI: 10.1109/TSE.2017.2754374.

[37] M. Kim, T. Zimmermann, R. DeLine and A. Begel, "The Emerging Role of Data Scientists on Software Development Teams", in *Proceedings of the 38th International Conference on Software Engineering*, New York, NY, USA: ACM, 2016, pp. 96–107, ISBN: 978-1-4503-3900-1. DOI: 10.1145/2884781.2884783. [Online]. Available: https://doi.acm.org/10.1145/2884781.2884783.

[38] D. Sculley et al., "Hidden Technical Debt in Machine Learning Systems", in *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2*, ser. NIPS'15, Montreal, Canada: MIT Press, 2015, pp. 2503–2511.

[39] L. E. Lwakatare, A. Raj, J. Bosch, H. H. Olsson and I. Crnkovic, "A Taxonomy of Software Engineering Challenges for Machine Learning Systems: An Empirical Investigation", in *Agile Processes in Software Engineering and Extreme Programming*, P. Kruchten, S. Fraser and F. Coallier, Eds., Cham: Springer International Publishing, 2019, pp. 227–243, ISBN: 978-3-030-19034-7.

[40] E. Nascimento, A. Nguyen-Duc, I. Sundbø and T. Conte, *Software engineering for artificial intelligence and machine learning software: A systematic literature review*, 2020. arXiv: 2011.03751 [cs.SE].

[41] D. Soni, R. L. Nord and C. Hofmeister, "Software Architecture in Industrial Applications", in *Proceedings of the 17th International Conference on Software Engineering*, ser. ICSE '95, Seattle, Washington, USA: Association for Computing Machinery, 1995, pp. 196–207, ISBN: 0897917081. DOI: 10.1145/225014.225033. [Online]. Available: https://doi.org/10.1145/225014.225033.

[42] F. Bachmann et al., "Software Architecture Documentation in Practice: Documenting Architectural Layers", Jan. 2000.

[43] A. Eden and R. Kazman, "Architecture, Design, Implementation.", Jan. 2003, pp. 149–159. DOI: 10.1109/ICSE.2003.1201196.

[44] P. Winder, *Reinforcement Learning, Industrial Applications of Intelligent Agents*. O'Reilly, 2021.

[45] Y. LeCun, Y. Bengio and G. Hinton, "Deep learning", *nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[46] S. Pouyanfar et al., "A survey on deep learning: Algorithms, techniques, and applications", *ACM computing surveys (CSUR)*, vol. 51, no. 5, pp. 1–36, 2018.

[47] Z. Zhu, K. Lin, A. K. Jain and J. Zhou, "Transfer Learning in Deep Reinforcement Learning: A Survey", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 45, no. 11, pp. 13 344–13 362, 2023. DOI: 10.1109/TPAMI.2023.3292075.

[48] T. Islam, D. M. H. Abid, T. Rahman, Z. Zaman, K. Mia and R. Hossain, "Transfer Learning in Deep Reinforcement Learning", in *Proceedings of Seventh International Congress on Information and Communication Technology*, X.-S. Yang, S. Sherratt, N. Dey and A. Joshi, Eds., Singapore: Springer Nature Singapore, 2023, pp. 145–153, ISBN: 978-981-19-1607-6.

[49] C. Tan, F. Sun, T. Kong, W. Zhang, C. Yang and C. Liu, "A survey on deep transfer learning", in *International Conference on Artificial Neural Networks*, Springer, 2018, pp. 270–279.

[50] S. Chen et al., *Self-Ensemble Protection: Training Checkpoints Are Good Data Protectors*, 2023. arXiv: 2211.12005 [cs.LG].

[51] A. Eisenman et al., "Check-n-run: A checkpointing system for training recommendation models", *arXiv preprint arXiv:2010.08679*, vol. 5, 2020.

[52] A. Hosna, E. Merry, J. Gyalmo, Z. Alom, Z. Aung and M. A. Azim, "Transfer learning: a friendly introduction", *Journal of Big Data*, vol. 9, no. 1, p. 102, 2022.

[53] P. Kriens and T. Verbelen, "Software Engineering Practices for Machine Learning", *CoRR*, vol. abs/1906.10366, 2019. arXiv: 1906.10366. [Online]. Available: http://arxiv.org/abs/1906.10366.

[54] S. Amershi et al., "Software Engineering for Machine Learning: A Case Study", *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pp. 291–300, 2019. [Online]. Available: https://api.semanticscholar.org/CorpusID:88499204.

[55] M. S. Rahman, E. Rivera, F. Khomh, Y. Guéhéneuc and B. Lehnert, "Machine Learning Software Engineering in Practice: An Industrial Case Study", *CoRR*, vol. abs/1906.07154, 2019. arXiv: 1906.07154. [Online]. Available: http://arxiv.org/abs/1906.07154.

[56] L. Faubel and K. Schmid, "An Analysis of MLOps Practices", Software Systems Engineering, Institut für Informatik, Universität Hildesheim, - Universitätsplatz 1, 31134 Hildesheim, Hildesheimer Informatik-Berichte 1/2023, SSE 1/23/E, 2023.

[57] M. M. John, H. H. Olsson and J. Bosch, "Towards MLOps: A framework and maturity model", in *2021 47th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, IEEE, 2021, pp. 1–8.

[58] E. Bisong et al., *Building machine learning and deep learning models on Google cloud platform*. Springer, 2019.

[59] M. Steidl, M. Felderer and R. Ramler, "The pipeline for the continuous development of artificial intelligence models—Current state of research and practice", *Journal of Systems and Software*, vol. 199, p. 111 615, 2023.

[60] N. Polyzotis, S. Roy, S. E. Whang and M. Zinkevich, "Data management challenges in production machine learning", in *Proceedings of the 2017 ACM International Conference on Management of Data*, 2017, pp. 1723–1726.

[61] M. Bernas, A. Mykytyshyn, V. Kartashov, V. Levytskyi and D. Martjanov, "The Role of Cyber-Physical Systems and Internet of Things in Development Smart Cities for Industry 4.0", in *Congreso Internacional de Tecnologías e Innovación*, 2023. [Online]. Available: https://api.semanticscholar.org/CorpusID:261698719.

[62] J. Leest, I. Gerostathopoulos and C. Raibulet, "Evolvability of Machine Learning-based Systems: An Architectural Design Decision Framework", in *2023 IEEE 20th International Conference on Software Architecture Companion (ICSA-C)*, 2023, pp. 106–110. DOI: 10.1109/ICSA-C57050.2023.00033.

[63] R. Dhar, K. Vaidhyanathan and V. Varma, "Can LLMs Generate Architectural Design Decisions? - An Exploratory Empirical Study", in *2024 IEEE 21st International Conference on Software Architecture (ICSA)*, 2024, pp. 79–89. DOI: 10.1109/ICSA59870.2024.00016.

[64] A. Fischbach et al., "CAAI—a cognitive architecture to introduce artificial intelligence in cyber-physical production systems", *The International Journal of Advanced Manufacturing Technology*, vol. 111, pp. 1–18, Nov. 2020. DOI: 10.1007/s00170-020-06094-z.

[65] J. Corbin and A. L. Strauss, "Grounded theory research: Procedures, canons, and evaluative criteria", *Qualitative Sociology*, vol. 13, pp. 3–20, 1 1990.

[66] S. Cook et al., "Unified Modeling Language (UML) Version 2.5.1", Object Management Group (OMG), Standard, Dec. 2017. [Online]. Available: https://www.omg.org/spec/UML/2.5.1.

[67] J. DeFranco and P. Laplante, "A content analysis process for qualitative software engineering research", *Innovations in Systems and Software Engineering*, vol. 13, pp. 1–13, Sep. 2017. DOI: 10.1007/s11334-017-0287-0.

[68] E. Ntentos, S. J. Warnett and U. Zdun, "Rule-Based Assessment of Reinforcement Learning Practices Using Large Language Models", in *2025 IEEE/ACM 4th International Conference on AI Engineering – Software Engineering for AI (CAIN)*, 2025, pp. 1–11. DOI: 10.1109/CAIN66642.2025.00009.

[69] S. J. Warnett and U. Zdun, "On the Understandability of MLOps System Architectures", *IEEE Transactions on Software Engineering*, vol. 50, no. 5, pp. 1015–1039, 2024. DOI: 10.1109/TSE.2024.3367488.

[70] E. Ntentos, S. J. Warnett and U. Zdun, "On the Understandability of Machine Learning Practices in Deep Learning and Reinforcement Learning Based Systems", *Journal of Systems and Software*, vol. 222, p. 112 343, 2025, ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2025.112343. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121225000111.

[71] S. J. Warnett, E. Ntentos and U. Zdun, "A Model-Driven, Metrics-Based Approach to Assessing Support for Quality Aspects in MLOps System Architectures", *Journal of Systems and Software*, vol. 220, p. 112 257, 2025, ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2024.112257. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121224003017.

[72] S. J. Warnett, E. Ntentos and U. Zdun, "MLOps Pipeline Generation for Reinforcement Learning: A Low-Code Approach Using Large Language Models", *Journal of Systems and Software*, vol. 235, p. 112 760, 2026, ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2025.112760. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121225004297.

[73] S. J. Warnett, U. Zdun and S. Geiger, "RLOps Pipeline Development with Low-Code and Large Language Models for Industry 4.0", in *CAIN 2026 - 5th International Conference on AI Engineering - Software Engineering for AI*, Jan. 2026. DOI: https://doi.org/10.1145/3793653.3793779. [Online]. Available: http://eprints.cs.univie.ac.at/8676/.

[74] S. J. Warnett and U. Zdun, "AI-Powered Architecting for Industry 4.0 Cyber-Physical Production Systems: A Novel Approach, Research Problems and Challenges", in *Software Architecture. ECSA 2025 Tracks and Workshops*, D. Bianculli et al., Eds., Cham: Springer Nature Switzerland, 2026, pp. 155–168, ISBN: 978-3-032-04403-7. DOI: https://doi.org/10.1007/978-3-032-04403-7_15.

[75] E. Ntentos, A. Amiri, S. Warnett and U. Zdun, "Decision-Making Support for Data Integration in Cyber-Physical-System Architectures", in *Service-Oriented Computing*, F. Monti, S. Rinderle-Ma, A. Ruiz Cortés, Z. Zheng and M. Mecella, Eds., Cham: Springer Nature Switzerland, 2023, pp. 137–152, ISBN: 978-3-031-48421-6. DOI: https://doi.org/10.1007/978-3-031-48421-6_10.

[76] B. G. Glaser and A. L. Strauss, *The Discovery of Grounded Theory: Strategies for Qualitative Research*. de Gruyter, 1967.

[77] Python Software Foundation, *Python Language Reference*, Version 3.14.3, 2026. [Online]. Available: https://docs.python.org/3/reference/index.html.

[78] U. Zdun, *Codeable Models*, https://github.com/uzdun/CodeableModels.

[79] PlantUML Team, *PlantUML*, PlantUML. [Online]. Available: https://plantuml.com.

[80] A. R. Hevner, S. T. March, J. Park and S. Ram, "Design science in information systems research", *MIS quarterly*, pp. 75–105, 2004.

[81] S. Easterbrook, J. Singer, M.-A. Storey and D. Damian, "Selecting empirical methods for software engineering research", in *Guide to advanced empirical software engineering*, Springer, 2008, pp. 285–311.

[82] G. Charness, U. Gneezy and M. A. Kuhn, "Experimental methods: Between-subject and within-subject design", *Journal of Economic Behavior & Organization*, vol. 81, no. 1, pp. 1–8, 2012, ISSN: 0167-2681. DOI: https://doi.org/10.1016/j.jebo.2011.08.009. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167268111002289.

[83] A. Jedlitschka, M. Ciolkowski and D. Pfahl, "Reporting Experiments in Software Engineering", in *Guide to Advanced Empirical Software Engineering*, F. Shull, J. Singer and D. I. K. Sjøberg, Eds. London: Springer London, 2008, pp. 201–228, ISBN: 978-1-84800-044-5. DOI: 10.1007/978-1-84800-044-5_8. [Online]. Available: https://doi.org/10.1007/978-1-84800-044-5_8.

[84] B. Kitchenham et al., "Preliminary Guidelines for Empirical Research in Software Engineering", *Software Engineering, IEEE Transactions on*, vol. 28, pp. 721–734, Sep. 2002. DOI: 10.1109/TSE.2002.1027796.

[85] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell and A. Wesslén, *Experimentation in Software Engineering*. Springer Publishing Company, Incorporated, 2012, ISBN: 3642290434.

[86] N. Juristo and A. Moreno, *Basics of Software Engineering Experimentation*. Jan. 2001, ISBN: 978-0-7923-7990-4. DOI: 10.1007/978-1-4757-3304-4.

[87] B. Kitchenham et al., "Robust Statistical Methods for Empirical Software Engineering", *Empirical Softw. Engg.*, vol. 22, no. 2, pp. 579–630, Apr. 2017, ISSN: 1382-3256. DOI: 10.1007/s10664-016-9437-5. [Online]. Available: https://doi.org/10.1007/s10664-016-9437-5.

[88] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering", *Empirical Software Engineering*, vol. 14, pp. 131–164, 2009. [Online]. Available: https://api.semanticscholar.org/CorpusID:207144526.

[89] R. K. Yin, *Case study research and applications : design and methods*, eng, Sixth. Los Angeles London New Delhi Singapore Washington, DC Melbourne: SAGE, 2018, ISBN: 9781506336169.

[90] A. Bucaioni, M. Weyssow, J. He, Y. Lyu and D. Lo, "Artificial Intelligence for Software Architecture: Literature Review and the Road Ahead", in *2030 Software Engineering - 2025*, Jun. 2025. [Online]. Available: http://www.ipr.mdu.se/publications/7175-.

[91] X. Zhou et al., *Using LLMs in Generating Design Rationale for Software Architecture Decisions*, 2025. arXiv: 2504.20781 [cs.SE]. [Online]. Available: https://arxiv.org/abs/2504.20781.

[92] O. Gheibi, D. Weyns and F. Quin, "Applying Machine Learning in Self-adaptive Systems: A Systematic Literature Review", *ACM Trans. Auton. Adapt. Syst.*, vol. 15, no. 3, Aug. 2021, ISSN: 1556-4665. DOI: 10.1145/3469440.

[93] A. Ramic and S. Kugele, *A Systematic Mapping Study on Software Architecture for AI-based Mobility Systems*, 2025. arXiv: 2506.01595 [cs.SE]. [Online]. Available: https://arxiv.org/abs/2506.01595.

[94] M. I. Jordan and T. M. Mitchell, "Machine learning: Trends, perspectives, and prospects", *Science*, vol. 349, no. 6245, pp. 255–260, 2015.

[95] V. Garousi, M. Felderer, M. V. Mäntylä and A. Rainer, "Benefitting from the Grey Literature in Software Engineering Research", in *Contemporary Empirical Methods in Software Engineering*, M. Felderer and G. H. Travassos, Eds. Cham: Springer International Publishing, 2020, pp. 385–413, ISBN: 978-3-030-32489-6. DOI: 10.1007/978-3-030-32489-6_14. [Online]. Available: https://doi.org/10.1007/978-3-030-32489-6_14.

[96] A. L. Strauss and J. M. Corbin, *Basics of qualitative research: techniques and procedures for developing grounded theory*. Sage Publications, Thousand Oaks, Calif, 1998, XIII, 312 s.

[97] V. Garousi, M. Felderer and M. V. Mäntylä, "Guidelines for including grey literature and conducting multivocal literature reviews in software engineering", *Inf. Softw. Technol.*, vol. 106, pp. 101–121, 2019. DOI: 10.1016/j.infsof.2018.09.006.

[98] A. Singjai, U. Zdun and O. Zimmermann, "Practitioner Views on the Interrelation of Microservice APIs and Domain-Driven Design: A Grey Literature Study Based on Grounded Theory", in *2021 IEEE 18th International Conference on Software Architecture (ICSA)*, 2021, pp. 25–35. DOI: 10.1109/ICSA51549.2021.00011.

[99] A. Singjai, G. Simhandl and U. Zdun, "On the practitioners' understanding of coupling smells — A grey literature based Grounded-Theory study", *Information and Software Technology*, vol. 134, p. 106 539, 2021, ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2021.106539. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584921000264.

[100] R. B. Johnson and L. Christensen, *Educational research: Quantitative, qualitative, and mixed approaches*. SAGE Publications, Incorporated, 2019.

[101] A. Rainer and A. Williams, "Using blog-like documents to investigate software practice: benefits, challenges and research directions", English, *Journal of Software: Evolution and Process*, Aug. 2019, ISSN: 2047-7481. DOI: 10.1002/smr.2197.

[102] L. Lamport, *LaTeX - A Document Preparation System*, eng. Addison-Wesley, 1986, ISBN: 0-201-15790-X.

[103] S. J. Warnett and U. Zdun, *Architectural Design Decisions for the Machine Learning Workflow: Dataset and Code*, Zenodo, https://doi.org/10.5281/zenodo.5730291, Nov. 2021. DOI: 10.5281/zenodo.5730291.

[104] H. Washizaki, H. Uchida, F. Khomh and Y.-G. Guéhéneuc, "Studying Software Engineering Patterns for Designing Machine Learning Systems", in *2019 10th International Workshop on Empirical Software Engineering in Practice (IWESEP)*, 2019, pp. 49–495. DOI: 10.1109/IWESEP49350.2019.00017.

[105] J. Bosch, H. Olsson and I. Crnkovic, "Engineering AI Systems: A Research Agenda", in Jan. 2021, pp. 1–19, ISBN: 9781799851028. DOI: 10.4018/978-1-7998-5101-1.ch001.

[106] A. Serban, K. van der Blom, H. Hoos and J. Visser, "Adoption and Effects of Software Engineering Best Practices in Machine Learning", *Proceedings of the 14th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, Oct. 2020. DOI: 10.1145/3382494.3410681. [Online]. Available: http://dx.doi.org/10.1145/3382494.3410681.

[107] S. Martínez-Fernández et al., *Software Engineering for AI-Based Systems: A Survey*, 2021. arXiv: 2105.01984 [cs.SE].

[108] I. Alves, L. Alexandre Ferreira Leite, P. Meirelles and C. Silva Rocha Aguiar, *Product Engineering for Machine Learning: A Grey Literature Review*, 2020. [Online]. Available: https://conf.researchr.org/details/wain-2021/wain-2021-papers/31/Product-Engineering-for-Machine-Learning-A-Grey-Literature-Review.

[109] W. Chang and N. Grady, *NIST Big Data Interoperability Framework: Volume 1, Definitions*, en, Oct. 2019. DOI: https://doi.org/10.6028/NIST.SP.1500-1r2.

[110] K. Charmaz, *Constructing grounded theory: a practical guide through qualitative analysis*, English. London; Thousand Oaks, Calif.: Sage Publications, 2006, ISBN: 9780761973539.

[111] S. Mäkinen, H. Skogström, E. Laaksonen and T. Mikkonen, "Who Needs MLOps: What Data Scientists Seek to Accomplish and How Can MLOps Help?", in *2021 IEEE/ACM 1st Workshop on AI Engineering - Software Engineering for AI (WAIN)*, 2021, pp. 109–112. DOI: 10.1109/WAIN52551.2021.00024.

[112] S. J. Warnett and U. Zdun, *Architectural Design Decisions for Machine Learning Deployment: Dataset and Code*, Jan. 2022. DOI: 10.5281/zenodo.5823458. [Online]. Available: https://doi.org/10.5281/zenodo.5823458.

[113] K.-J. Stol, P. Ralph and B. Fitzgerald, "Grounded theory in software engineering research: a critical review and guidelines", in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16, Austin, Texas: Association for Computing Machinery, 2016, pp. 120–131, ISBN: 9781450339001. DOI: 10.1145/2884781.2884833. [Online]. Available: https://doi.org/10.1145/2884781.2884833.

[114] C. Hagstrom, S. Kendall and H. Cunningham, "Googling for grey: Using Google and Duckduckgo to find grey literature", in *23rd Cochrane Colloquium. Cochrane database systematic reviews supplements*, 2015, pp. 1–327.

[115] J. Piasecki, M. Waligora and V. Dranseika, "Google search as an additional source in systematic reviews", *Science and engineering ethics*, vol. 24, no. 2, pp. 809–810, 2018.

[116] R. Suddaby, "From the Editors: What Grounded Theory is Not", *Academy of Management Journal*, vol. 49, no. 4, pp. 633–642, 2006. DOI: 10.5465/amj.2006.22083020. eprint: https://doi.org/10.5465/amj.2006.22083020. [Online]. Available: https://doi.org/10.5465/amj.2006.22083020.

[117] D. Lee, N. He, P. Kamalaruban and V. Cevher, "Optimization for reinforcement learning: From a single agent to cooperative agents", *IEEE Signal Processing Magazine*, vol. 37, no. 3, pp. 123–135, 2020.

[118] L. Canese et al., "Multi-agent reinforcement learning: A review of challenges and applications", *Applied Sciences*, vol. 11, no. 11, p. 4948, 2021.

[119] E. Liang et al., "RLlib: Abstractions for distributed reinforcement learning", in *International conference on machine learning*, PMLR, 2018, pp. 3053–3062.

[120] J. Coplien, *Software Patterns: Management Briefings*. SIGS, New York, 1996.

[121] R. R. Eimer T. Lindauer M., "Hyperparameters in Reinforcement Learning and How to Tune Them", in *International Conference on Machine Learning*, 2023, pp. 9104–9149.

[122] H. Washizaki et al., "Software-Engineering Design Patterns for Machine Learning Applications", *Computer*, vol. 55, no. 3, pp. 30–39, 2022. DOI: 10.1109/MC.2021.3137227.

[123] R. Sharma and K. Davuluri, "Design patterns for Machine Learning Applications", in *2019 3rd International Conference on Computing Methodologies and Communication (ICCMC)*, 2019, pp. 818–821. DOI: 10.1109/ICCMC.2019.8819692.

[124] O. Zimmermann, J. Koehler, F. Leymann, R. Polley and N. Schuster, "Managing Architectural Decision Models with Dependency Relations, Integrity Constraints, and Production Rules", *J. Syst. Softw.*, vol. 82, no. 8, pp. 1249–1267, 2009.

[125] I. Lytra, S. Sobernig and U. Zdun, "Architectural Decision Making for Service-Based Platform Integration: A Qualitative Multi-Method Study", in *Proc. of WICSA/ECSA*, 2012.

[126] C. Pautasso, O. Zimmermann and F. Leymann, "RESTful Web Services vs. Big Web Services: Making the Right Architectural Decision", in *Proc. of the 17th World Wide Web Conference*, 2008, pp. 805–814.

[127] I. Gorton, J. Klein and A. Nurgaliev, "Architecture Knowledge for Evaluating Scalable Databases", in *Proc. of the 12th Working IEEE/IFIP Conference on Software Architecture*, 2015, pp. 95–104.

[128] U. van Heesch, P. Avgeriou and R. Hilliard, "A documentation framework for architecture decisions", *J. Syst. Softw.*, vol. 85, no. 4, pp. 795–820, 2012.

[129] U. Zdun, M. Stocker, O. Zimmermann, C. Pautasso and D. Lübke, "Supporting Architectural Decision Making on Quality Aspects of Microservice APIs", in *16th International Conference on Service-Oriented Computing*, Springer, 2018.

[130] C. Hentrich, U. Zdun, V. Hlupic and F. Dotsika, "An Approach for Pattern Mining Through Grounded Theory Techniques and Its Applications to Process-driven SOA Patterns", in *Proc. of the 18th European Conference on Pattern Languages of Program*, Irsee, Germany, 2015, 9:1–9:16.

[131] E. Ntentos, S. J. Warnett and U. Zdun, *Supporting Architectural Decision Making on Training Strategies in Reinforcement Learning Architectures*, Feb. 2024. DOI: 10.5281/zenodo.10624377. [Online]. Available: https://doi.org/10.5281/zenodo.10624377.

[132] L. Busoniu, R. Babuska and B. De Schutter, "A comprehensive survey of multiagent reinforcement learning", *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, vol. 38, no. 2, pp. 156–172, 2008.

[133] J. Orr and A. Dutta, "Multi-Agent Deep Reinforcement Learning for Multi-Robot Applications: A Survey", *Sensors*, vol. 23, 2023.

[134] M. R. Samsami and H. Alimadad, "Distributed deep reinforcement learning: An overview", *arXiv preprint arXiv:2011.11012*, 2020.

[135] J. Yang, B. Zhu, J. Chen and Y.-G. Jiang, *Actor-critic for continuous action chunks: A reinforcement learning framework for long-horizon robotic manipulation with sparse reward*, 2025. arXiv: 2508.11143 [cs.RO]. [Online]. Available: https://arxiv.org/abs/2508.11143.

[136] J. Schulman, F. Wolski, P. Dhariwal, A. Radford and O. Klimov, *Proximal policy optimization algorithms*, 2017. arXiv: 1707.06347 [cs.LG]. [Online]. Available: https://arxiv.org/abs/1707.06347.

[137] J. Schulman, S. Levine, P. Abbeel, M. Jordan and P. Moritz, "Trust region policy optimization", in *Proceedings of the 32nd International Conference on Machine Learning*, F. Bach and D. Blei, Eds., ser. Proceedings of Machine Learning Research, vol. 37, Lille, France: PMLR, Jul. 2015, pp. 1889–1897. [Online]. Available: https://proceedings.mlr.press/v37/schulman15.html.

[138] Martín Abadi et al., *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*, Software available from tensorflow.org, 2015. [Online]. Available: https://www.tensorflow.org/.

[139] C. Paduraru et al., "Challenges of real-world reinforcement learning: definitions, benchmarks and analysis", *Machine Learning*, vol. 110, pp. 2419–2468, 2021. [Online]. Available: https://api.semanticscholar.org/CorpusID:234868359.

[140] S. Stevanetic, K. Plakidas, T. B. Ionescu, F. Li, D. Schall and U. Zdun, "Tool Support for the Architectural Design Decisions in Software Ecosystems", *Proceedings of the 2015 European Conference on Software Architecture Workshops*, 2015. [Online]. Available: https://api.semanticscholar.org/CorpusID:1144013.

[141] J. Hao et al., "Exploration in deep reinforcement learning: From single-agent to multiagent domain", *IEEE Transactions on Neural Networks and Learning Systems*, 2023.

[142] H. M. Schwartz, *Multi-agent machine learning: A reinforcement approach*. John Wiley & Sons, 2014.

[143] K. Zhang, Z. Yang and T. Başar, "Multi-agent reinforcement learning: A selective overview of theories and algorithms", *Handbook of reinforcement learning and control*, pp. 321–384, 2021.

[144] N. Liyanawaduge, E. Kumarasinghe, S. S. Iyer, A. K. Kulatunga and G. Lakmal, "Digital Twin & Virtual Reality Enabled Conveyor System to Promote Learning Factory Concept", *2023 IEEE 17th International Conference on Industrial and Information Systems (ICIIS)*, pp. 85–90, 2023. [Online]. Available: https://api.semanticscholar.org/CorpusID:262076517.

[145] V. Havard, B. Jeanne, M. Lacomblez and D. Baudry, "Digital twin and virtual reality: a co-simulation environment for design and assessment of industrial workstations", *Production & Manufacturing Research*, vol. 7, pp. 472–489, 2019. [Online]. Available: https://api.semanticscholar.org/CorpusID:202789864.

[146] A. del Real Torres, D. S. Andreiana, Á. Ojeda Roldán, A. Hernández Bustos and L. E. Acevedo Galicia, "A Review of Deep Reinforcement Learning Approaches for Smart Manufacturing in Industry 4.0 and 5.0 Framework", *Applied Sciences*, vol. 12, no. 23, 2022, ISSN: 2076-3417. DOI: 10.3390/app122312377. [Online]. Available: https://www.mdpi.com/2076-3417/12/23/12377.

[147] European Commission and Directorate-General for Research and Innovation, M. Breque, L. De Nul and A. Petridis, *Industry 5.0 – Towards a sustainable, human-centric and resilient European industry*. Publications Office of the European Union, 2021. DOI: doi/10.2777/308407.

[148] T. Kegyes, Z. Süle and J. Abonyi, "The Applicability of Reinforcement Learning Methods in the Development of Industry 4.0 Applications", *Complex.*, vol. 2021, 7179374:1–7179374:31, 2021. DOI: 10.1155/2021/7179374. [Online]. Available: https://api.semanticscholar.org/CorpusID:246751425.

[149] S. J. Warnett and U. Zdun, *Bridging the Gap Between MLOps and RLOps: An Industry 4.0 Case Study on Architectural Design Decisions in Practice: Replication Package*, Jan. 2025. DOI: 10.5281/zenodo.14722767. [Online]. Available: https://doi.org/10.5281/zenodo.14722767.

[150] S. Levine, A. Kumar, G. Tucker and J. Fu, "Offline Reinforcement Learning: Tutorial, Review, and Perspectives on Open Problems", *ArXiv*, vol. abs/2005.01643, 2020. [Online]. Available: https://api.semanticscholar.org/CorpusID:218486979.

[151] J. Parker-Holder et al., "Automated Reinforcement Learning (AutoRL): A Survey and Open Problems", *J. Artif. Int. Res.*, vol. 74, Sep. 2022, ISSN: 1076-9757. DOI: 10.1613/jair.1.13596. [Online]. Available: https://doi.org/10.1613/jair.1.13596.

[152] X. Wang, B. Li, Y. Zhang, B. Kailkhura and K. Nahrstedt, "Robusta: Robust AutoML for Feature Selection via Reinforcement Learning", *ArXiv*, vol. abs/2101.05-950, 2021. [Online]. Available: https://api.semanticscholar.org/CorpusID:231627788.

[153] E. Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley, 2003.

[154] M. Völter, "Architecture as Language", *Software, IEEE*, vol. 27, pp. 56–64, Mar. 2010. DOI: 10.1109/MS.2010.38.

[155] B. Rumpe and R. France, *Variability in UML language and semantics*, 2011.

[156] J. Whittle, "Formal approaches to systems analysis using UML: an overview", *Advanced Topics in Database Research, Volume 1*, pp. 324–341, 2002.

[157] R. France, A. Evans, K. Lano and B. Rumpe, "The UML as a formal modeling notation", *Computer Standards & Interfaces*, vol. 19, no. 7, pp. 325–334, 1998, ISSN: 0920-5489. DOI: https://doi.org/10.1016/S0920-5489(98)00020-8. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0920548998000208.

[158] C. Czepa and U. Zdun, "How Understandable Are Pattern-based Behavioral Constraints for Novice Software Designers?", *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 28, pp. 1–38, 2019.

[159] C. Czepa and U. Zdun, "On the Understandability of Temporal Properties Formalized in Linear Temporal Logic, Property Specification Patterns and Event Processing Language", *IEEE Transactions on Software Engineering*, vol. 46, pp. 100–112, 2020.

[160] P. Paulweber, G. Simhandl and U. Zdun, "On the Understandability of Language Constructs to Structure the State and Behavior in Abstract State Machine Specifications: A Controlled Experiment", *J. Syst. Softw.*, vol. 178, p. 110 987, 2021.

[161] R. Solingen, V. Basili, G. Caldiera and D. Rombach, "Goal Question Metric (GQM) Approach", in Jan. 2002, ISBN: 9780471028956. DOI: 10.1002/0471028959.sof142.

[162] Q. Yao et al., *Taking Human out of Learning Applications: A Survey on Automated Machine Learning*, 2019. arXiv: 1810.13306 [cs.AI].

[163] M.-A. Zöller and M. F. Huber, *Benchmark and Survey of Automated Machine Learning Frameworks*, 2021. arXiv: 1904.12054 [cs.LG].

[164] U. Zdun, E. Ntentos, K. Plakidas, A. E. Malki, D. Schall and F. Li, "On the Design and Architecture of Deployment Pipelines in Cloud- and Service-Based Computing - A Model-Based Qualitative Study", 2019, pp. 141–145. [Online]. Available: https://api.semanticscholar.org/CorpusID:201810286.

[165] M. Shahin, M. Zahedi, M. A. Babar and L. Zhu, "An empirical study of architecting for continuous delivery and deployment", *Empirical Softw. Engg.*, vol. 24, no. 3, pp. 1061–1108, Jun. 2019, ISSN: 1382-3256. DOI: 10.1007/s10664-018-9651-4. [Online]. Available: https://doi.org/10.1007/s10664-018-9651-4.

[166] G. Schermann, J. Cito, P. Leitner, U. Zdun and H. Gall, "An empirical study on principles and practices of continuous delivery and deployment", PeerJ Preprints, Tech. Rep., 2016.

[167] D. A. Tamburri, "Sustainable MLOps: Trends and Challenges", *2020 22nd International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*, pp. 17–23, 2020. [Online]. Available: https://api.semanticscholar.org/CorpusID:232063132.

[168] A. Paleyes, R.-G. Urma and N. D. Lawrence, "Challenges in Deploying Machine Learning: A Survey of Case Studies", *ACM Computing Surveys*, vol. 55, pp. 1–29, 2020. [Online]. Available: https://api.semanticscholar.org/CorpusID:227053929.

[169] L. Allodi, M. Cremonini, F. Massacci and W. Shim, "Measuring the accuracy of software vulnerability assessments: experiments with students and professionals", English, *Empirical Software Engineering*, vol. 25, no. 2, pp. 1063–1094, Mar. 2020, ISSN: 1382-3256. DOI: 10.1007/s10664-019-09797-4.

[170] L. Allodi, S. Biagioni, B. Crispo, K. Labunets, F. Massacci and W. Santos, "Estimating the Assessment Difficulty of CVSS Environmental Metrics: An Experiment", in *Future Data and Security Engineering*, T. K. Dang, R. Wagner, J. Küng, N. Thoai, M. Takizawa and E. J. Neuhold, Eds., Cham: Springer International Publishing, 2017, pp. 23–39, ISBN: 978-3-319-70004-5.

[171] K. Labunets, F. Paci, F. Massacci and R. Ruprai, "An Experiment on Comparing Textual vs. Visual Industrial Methods for Security Risk Assessment", Aug. 2014. DOI: 10.1109/EmpiRE.2014.6890113.

[172] Z. Sharafi, A. Marchetto, A. Susi, G. Antoniol and Y.-G. Guéhéneuc, "An empirical study on the efficiency of graphical vs. textual representations in requirements comprehension", May 2013, pp. 33–42. DOI: 10.1109/ICPC.2013.6613831.

[173] R Core Team, *R: A Language and Environment for Statistical Computing*, R Foundation for Statistical Computing, Vienna, Austria, 2021. [Online]. Available: https://www.R-project.org/.

[174] W. Heijstek, T. Kühne and M. R. V. Chaudron, "Experimental Analysis of Textual and Graphical Representations for Software Architecture Design", *2011 International Symposium on Empirical Software Engineering and Measurement*, pp. 167–176, 2011.

[175] R. Jolak et al., "Software Engineering Whispers: The Effect of Textual vs. Graphical Software Design Descriptions on Software Design Communication", *Empirical Softw. Engg.*, vol. 25, no. 6, pp. 4427–4471, Nov. 2020, ISSN: 1382-3256. DOI: 10.1007/s10664-020-09835-6. [Online]. Available: https://doi.org/10.1007/s10664-020-09835-6.

[176] M. Pradella, M. Rossi and D. Mandrioli, "A UML-Compatible Formal Language for System Architecture Description", vol. 3530, Jun. 2005, pp. 234–246, ISBN: 978-3-540-26612-9. DOI: 10.1007/11506843_17.

[177] L. Lavazza, G. Quaroni and M. Venturelli, "Combining UML and formal notations for modelling real-time systems", in *Proceedings of the 8th European Software Engineering Conference Held Jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. ESEC/FSE-9, Vienna, Austria: Association for Computing Machinery, 2001, pp. 196–206, ISBN: 1581133901. DOI: 10.1145/503209.503236. [Online]. Available: https://doi.org/10.1145/503209.503236.

[178] M. Rodano and K. Giammarco, "A Formal Method for Evaluation of a Modeled System Architecture", *Procedia Computer Science*, vol. 20, pp. 210–215, 2013, Complex Adaptive Systems, ISSN: 1877-0509. DOI: https://doi.org/10.1016/j.procs.2013.09.263. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877050913010636.

[179] J. Li and J. Horgan, "Applying formal description techniques to software architectural design", *Computer Communications*, vol. 23, pp. 1169–1178, Jul. 2000. DOI: 10.1016/S0140-3664(99)00244-3.

[180] M. Höst, B. Regnell and C. Wohlin, "Using Students as Subjects - A Comparative Study of Students and Professionals in Lead-Time Impact Assessment", *Empirical Software Engineering*, vol. 5, pp. 201–214, Nov. 2000. DOI: 10.1023/A:1026586415054.

[181] Runeson, Per, "Using Students as Experiment Subjects – An Analysis on Graduate and Freshmen Student Data", eng, in *Proceedings 7th International Conference on Empirical Assessment & Evaluation in Software Engineering*, 2003, 95–102.

[182] M. Svahnberg, A. Aurum and C. Wohlin, "Using students as subjects - an empirical evaluation", in *International Symposium on Empirical Software Engineering and Measurement*, 2008.

[183] I. Salman, A. T. Misirli and N. Juristo, "Are Students Representatives of Professionals in Software Engineering Experiments?", vol. 1, pp. 666–676, 2015. DOI: 10.1109/ICSE.2015.82.

[184] D. Falessi et al., "Empirical software engineering experts on the use of students and professionals in experiments", *Empirical Softw. Engg.*, vol. 23, no. 1, pp. 452–489, Feb. 2018, ISSN: 1382-3256. DOI: 10.1007/s10664-017-9523-3. [Online]. Available: https://doi.org/10.1007/s10664-017-9523-3.

[185] S. J. Warnett and U. Zdun, *On the Understandability of MLOps System Architectures: Dataset and Code*, Zenodo, https://doi.org/10.5281/zenodo.7752176, Feb. 2024. DOI: 10.5281/zenodo.7752176.

[186] S. Stevanetic, M. A. Javed and U. Zdun, "Empirical evaluation of the understandability of architectural component diagrams", in *Proceedings of the WICSA 2014 Companion Volume*, ser. WICSA '14 Companion, Sydney, Australia: Association for Computing Machinery, 2014, ISBN: 9781450325233. DOI: 10.1145/2578128.2578230. [Online]. Available: https://doi.org/10.1145/2578128.2578230.

[187] T. Haitzer and U. Zdun, "Controlled experiment on the supportive effect of architectural component diagrams for design understanding of novice architects", in *Proceedings of the 7th European Conference on Software Architecture*, ser. ECSA'13, Montpellier, France: Springer-Verlag, 2013, pp. 54–71, ISBN: 9783642390302. DOI: 10.1007/978-3-642-39031-9_6. [Online]. Available: https://doi.org/10.1007/978-3-642-39031-9_6.

[188] J. Siegmund et al., "Do background colors improve program comprehension in the #ifdef hell?", *Empirical Software Engineering*, vol. 18, pp. 1–47, Aug. 2012. DOI: 10.1007/s10664-012-9208-x.

[189] B. Hoisl, S. Sobernig and M. Strembeck, "Comparing Three Notations for Defining Scenario-Based Model Tests: A Controlled Experiment", *Proceedings - 2014 9th International Conference on the Quality of Information and Communications Technology, QUATIC 2014*, pp. 95–104, Dec. 2014. DOI: 10.1109/QUATIC.2014.19.

[190] The Document Foundation, *LibreOffice*. [Online]. Available: https://www.libreoffice.org.

[191] Y. Shafranovich, *Common Format and MIME Type for Comma-Separated Values (CSV) Files*, RFC 4180, Oct. 2005. DOI: 10.17487/RFC4180. [Online]. Available: https://www.rfc-editor.org/info/rfc4180.

[192] M. S. Handcock, I. E. Fellows and K. J. Gile, *RDS: Respondent-Driven Sampling*, R package version 0.9-6, Los Angeles, CA, 2023. [Online]. Available: https://CRAN.R-project.org/package%20=%20RDS.

[193] S. S. Shapiro and M. B. Wilk, "An Analysis of Variance Test for Normality (Complete Samples)", *Biometrika*, vol. 52, pp. 591–611, 1965.

[194] T. W. Anderson and D. A. Darling, "A Test of Goodness of Fit", *Journal of the American Statistical Association*, vol. 49, no. 268, pp. 765–769, 1954. DOI: 10.1080/01621459.1954.10501232. eprint: https://www.tandfonline.com/doi/pdf/10.1080/01621459.1954.10501232. [Online]. Available: https://www.tandfonline.com/doi/abs/10.1080/01621459.1954.10501232.

[195] H. W. Lilliefors, "On the Kolmogorov-Smirnov Test for Normality with Mean and Variance Unknown", *Journal of the American Statistical Association*, vol. 62, pp. 399–402, 1967.

[196] A. N. Kolmogorov, "Sulla determinazione empirica di una legge didistribuzione", *Giorn Dell'inst Ital Degli Att*, vol. 4, pp. 89–91, 1933.

[197] N. Mohd Razali and B. Yap, "Power Comparisons of Shapiro-Wilk, Kolmogorov-Smirnov, Lilliefors and Anderson-Darling Tests", *J. Stat. Model. Analytics*, vol. 2, pp. 21–33, Jan. 2011.

[198] J. H. Bray and S. E. Maxwell, "Analyzing and Interpreting Significant MANOVAs", *Review of Educational Research*, vol. 52, no. 3, pp. 340–367, 1982, ISSN: 00346543, 19351046. Accessed: 13th Apr. 2023. [Online]. Available: http://www.jstor.org/stable/1170422.

[199] D. Bonett, "Robust Confidence Interval for a Ratio of Standard Deviations", *Applied Psychological Measurement - APPL PSYCHOL MEAS*, vol. 30, pp. 432–439, Sep. 2006. DOI: 10.1177/0146621605279551.

[200] H. Levene, "Robust tests for equality of variances", *Contributions to Probability and Statistics: Essays in Honor of Harold Hotelling*, I. Olkin and H. Hotelling, Eds., pp. 278–292, 1961.

[201] M. B. Brown and A. B. Forsythe, "Robust Tests for the Equality of Variances", *Journal of the American Statistical Association*, vol. 69, no. 346, pp. 364–367, 1974. DOI: 10.1080/01621459.1974.10482955. eprint: https://www.tandfonline.com/doi/pdf/10.1080/01621459.1974.10482955. [Online]. Available: https://www.tandfonline.com/doi/abs/10.1080/01621459.1974.10482955.

[202] W. H. Kruskal and W. A. Wallis, "Use of Ranks in One-Criterion Variance Analysis", *Journal of the American Statistical Association*, vol. 47, pp. 583–621, 1952.

[203] F. Wilcoxon, "Individual Comparisons by Ranking Methods", *Biometrics Bulletin*, vol. 1, no. 6, pp. 80–83, 1945, ISSN: 00994987. [Online]. Available: http://www.jstor.org/stable/3001968.

[204] D. W. Zimmerman, "Statistical Significance Levels of Nonparametric Tests Biased by Heterogeneous Variances of Treatment Groups", *The Journal of General Psychology*, vol. 127, no. 4, pp. 354–364, 2000, PMID: 11109998. DOI: 10.1080/00221300009598589. eprint: https://doi.org/10.1080/00221300009598589. [Online]. Available: https://doi.org/10.1080/00221300009598589.

[205] B. L. Welch, "The Generalization of "Student's" Problem when Several Different Population Variances are Involved", *Biometrika*, vol. 34, no. 1-2, pp. 28–35, Jan. 1947, ISSN: 0006-3444. DOI: 10.1093/biomet/34.1-2.28. eprint: https://academic.oup.com/biomet/article-pdf/34/1-2/28/553093/34-1-2-28.pdf. [Online]. Available: https://doi.org/10.1093/biomet/34.1-2.28.

[206] N. Cliff, "Dominance statistics: Ordinal analyses to answer ordinal questions.", *Psychological Bulletin*, vol. 114, pp. 494–509, 1993.

[207] H. D. Delaney and A. Vargha, "Comparing several robust tests of stochastic equality with ordinally scaled variables and small to moderate sized samples.", *Psychological methods*, vol. 7, pp. 485–503, 4 2002.

[208] L. M. Hsu, "Biases of success rate differences shown in binomial effect size displays.", *Psychological methods*, vol. 9 2, pp. 183–97, 2004.

[209] N. Cliff, "Answering Ordinal Questions with Ordinal Data Using Ordinal Statistics", *Multivariate Behavioral Research*, vol. 31, pp. 331–350, Jun. 2010. DOI: 10.1207/s15327906mbr3103_4.

[210] Y. Benjamini and Y. Hochberg, "Controlling The False Discovery Rate - A Practical And Powerful Approach To Multiple Testing", *J. Royal Statist. Soc., Series B*, vol. 57, pp. 289–300, Nov. 1995. DOI: 10.2307/2346101.

[211] C. Bonferroni, "Teoria statistica delle classi e calcolo delle probabilita", *Pubblicazioni del R Istituto Superiore di Scienze Economiche e Commerciali di Firenze*, vol. 8, pp. 3–62, 1936.

[212] O. J. Dunn, "Multiple Comparisons among Means", *Journal of the American Statistical Association*, vol. 56, pp. 52–64, 1961.

[213] E. Brunner and U. Munzel, "The Nonparametric Behrens-Fisher Problem: Asymptotic Theory and a Small-Sample Approximation", *Biometrical Journal*, vol. 42, pp. 17–25, 2000.

[214] K. Pearson, "Notes on regression and inheritance in the case of two parents", *Proceedings of the Royal Society of London*, vol. 58, pp. 240–242, 1895.

[215] M. G. Kendall, "A New Measure of Rank Correlation", *Biometrika*, vol. 30, no. 1-2, pp. 81–93, Jun. 1938, ISSN: 0006-3444. DOI: 10.1093/biomet/30.1-2.81. eprint: https://academic.oup.com/biomet/article-pdf/30/1-2/81/423380/30-1-2-81.pdf. [Online]. Available: https://doi.org/10.1093/biomet/30.1-2.81.

[216] C. Spearman, "The proof and measurement of association between two things. By C. Spearman, 1904.", *The American Journal of Psychology*, vol. 100 3-4, pp. 441–71, 1987.

[217] T. Cook and D. Campbell, *Quasi-experimentation: Design & Analysis Issues for Field Settings*. Houghton Mifflin, 1979, ISBN: 9780395307908. [Online]. Available: https://books.google.at/books?id=BFNqAAAAMAAJ.

[218] S. N. Haynes, D. Richard and E. S. Kubany, "Content validity in psychological assessment: A functional approach to concepts and methods.", *Psychological assessment*, vol. 7, no. 3, p. 238, 1995.

[219] A. Agrawal et al., *DynaQuant: Compressing Deep Learning Training Checkpoints via Dynamic Quantization*, 2023. arXiv: 2306.11800 [cs.LG].

[220] A. Farahani, B. Pourshojae, K. Rasheed and H. R. Arabnia, *A Concise Review of Transfer Learning*, 2021. arXiv: 2104.02144 [cs.LG].

[221] C. Raffel et al., *Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer*, 2023. arXiv: 1910.10683 [cs.LG].

[222] Y. Chen, Z. Liu, B. Ren and X. Jin, "On Efficient Constructions of Checkpoints", *CoRR*, vol. abs/2009.13003, 2020. arXiv: 2009.13003. [Online]. Available: https://arxiv.org/abs/2009.13003.

[223] A. Raffin, A. Hill, A. Gleave, A. Kanervisto, M. Ernestus and N. Dormann, "Stable-Baselines3: Reliable Reinforcement Learning Implementations", *Journal of Machine Learning Research*, vol. 22, no. 268, pp. 1–8, 2021. [Online]. Available: http://jmlr.org/papers/v22/20-1364.html.

[224] E. Ntentos, S. J. Warnett and U. Zdun, *On the Understandability of Machine Learning Practices in Deep Learning and Reinforcement Learning Based Systems*, Zenodo, Jan. 2025. DOI: 10.5281/zenodo.14677668. [Online]. Available: https://doi.org/10.5281/zenodo.14677668.

[225] J. N. Kather et al., "Multi-class texture analysis in colorectal cancer histology", *Scientific reports*, vol. 6, p. 27 988, 2016.

[226] K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition", in *International Conference on Learning Representations*, 2015.

[227] D. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization", *International Conference on Learning Representations*, Dec. 2014.

[228] D. Sweenor et al., *ML Ops: Operationalizing Data Science*. O'Reilly Media, Incorporated, 2020. [Online]. Available: https://books.google.at/books?id=StTvzQEACAAJ.

[229] C. Ebert, G. Gallardo, J. Hernantes and N. Serrano, "DevOps", *IEEE Software*, vol. 33, no. 3, pp. 94–100, 2016. DOI: 10.1109/MS.2016.68.

[230] S. Moreschi, G. Recupito, V. Lenarduzzi, F. Palomba, D. Hastbacka and D. Taibi, *Toward End-to-End MLOps Tools Map: A Preliminary Study based on a Multivocal Literature Review*, 2023. arXiv: 2304.03254 [cs.SE].

[231] A. Lima, L. Monteiro and A. Furtado, "MLOps: Practices, Maturity Models, Roles, Tools, and Challenges - A Systematic Literature Review", in *International Conference on Enterprise Information Systems*, 2022. [Online]. Available: https://api.semanticscholar.org/CorpusID:248718186.

[232] H. Washizaki, H. Uchida, F. Khomh and Y.-G. Guéhéneuc, "Machine learning architecture and design patterns", *IEEE Software*, vol. 8, 2020.

[233] L. Cardoso Silva et al., "Benchmarking Machine Learning Solutions in Production", in *2020 19th IEEE International Conference on Machine Learning and Applications (ICMLA)*, 2020, pp. 626–633. DOI: 10.1109/ICMLA51294.2020.00104.

[234] J. Zhou, A. H. Gandomi, F. Chen and A. Holzinger, "Evaluating the Quality of Machine Learning Explanations: A Survey on Methods and Metrics", *Electronics*, vol. 10, no. 5, 2021, ISSN: 2079-9292. DOI: 10.3390/electronics10050593. [Online]. Available: https://www.mdpi.com/2079-9292/10/5/593.

[235] D. V. Carvalho, E. M. Pereira and J. S. Cardoso, "Machine Learning Interpretability: A Survey on Methods and Metrics", *Electronics*, vol. 8, no. 8, 2019, ISSN: 2079-9292. DOI: 10.3390/electronics8080832. [Online]. Available: https://www.mdpi.com/2079-9292/8/8/832.

[236] S. J. Warnett and U. Zdun, *A Model-Driven, Metrics-Based Approach to Assessing Support for Quality Aspects in MLOps System Architectures: Replication Package*, Oct. 2024. DOI: 10.5281/zenodo.13941648. [Online]. Available: https://doi.org/10.5281/zenodo.13941648.

[237] J. Frank E. Harrell, *Regression Modeling Strategies: With Applications to Linear Models, Logistic and Ordinal Regression, and Survival Analysis*, 2nd. Springer, 2015.

[238] A. Airola, T. Pahikkala, W. Waegeman, B. De Baets and T. Salakoski, "An experimental comparison of cross-validation techniques for estimating the area under the ROC curve", *Computational Statistics & Data Analysis*, vol. 55, no. 4, pp. 1828–1844, 2011.

[239] G. W. Brier, "Verification of Forecasts Expressed in Terms of Probability", *Monthly Weather Review*, vol. 78, no. 1, pp. 1–3, 1950. DOI: https://doi.org/10.1175/1520-0493(1950)078<0001:VOFEIT>2.0.CO;2. [Online]. Available: https://journals.ametsoc.org/view/journals/mwre/78/1/1520-0493_1950_078_0001_vofeit_2_0_co_2.xml.

[240] M. Cowles and C. Davis, "On the Origins of the .05 Level of Statistical Significance", *American Psychologist*, vol. 37, pp. 553–558, May 1982. DOI: 10.1037/0003-066X.37.5.553.

[241] D. I. K. Sjøberg, B. Anda and A. Mockus, "Questioning software maintenance metrics: A comparative case study", in *Proceedings of the 2012 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, 2012, pp. 107–110. DOI: 10.1145/2372251.2372269.

[242] C. Watkins and P. Dayan, "Q-learning", *Machine Learning*, vol. 8, pp. 279–292, 1992. [Online]. Available: https://api.semanticscholar.org/CorpusID:208910339.

[243] V. Mnih, K. Kavukcuoglu, D. Silver et al., "Human-level control through deep reinforcement learning", *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[244] H. Chen, S. Lundberg and S.-I. Lee, "Checkpoint ensembles: Ensemble methods from a single training process", *arXiv preprint arXiv:1710.03282*, 2017.

[245] L. Li, K. Jamieson, G. DeSalvo, A. Rostamizadeh and A. Talwalkar, "Hyperband: A novel bandit-based approach to hyperparameter optimization", *Journal of Machine Learning Research*, vol. 18, no. 185, pp. 1–52, 2018.

[246] P. Hernandez-Leal, B. Kartal and M. E. Taylor, "A survey and critique of multiagent deep reinforcement learning", *Autonomous Agents and Multi-Agent Systems*, vol. 33, no. 6, pp. 750–797, 2019.

[247] S. Schneider, P.-J. Quéval, Á. Milánkovich, N. E. Díaz Ferreyra, U. Zdun and R. Scandariato, "Automatic Rule Checking for Microservices: Supporting Security Analysis with Explainability", *Available at SSRN 4658575*, 2023.

[248] E. Ntentos, S. J. Warnett and U. Zdun, *Rule-Based Assessment of Reinforcement Learning Practices Using Large Language Models*, Nov. 2024. DOI: 10.5281/zenodo.14050926. [Online]. Available: https://doi.org/10.5281/zenodo.14050926.

[249] T. Bray, *The JavaScript Object Notation (JSON) Data Interchange Format*, RFC 8259, Dec. 2017. DOI: 10.17487/RFC8259. [Online]. Available: https://www.rfc-editor.org/info/rfc8259.

[250] D. Silver et al., "A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play", *Science*, vol. 362, no. 6419, pp. 1140–1144, 2018. DOI: 10.1126/science.aar6404. eprint: https://www.science.org/doi/pdf/10.1126/science.aar6404. [Online]. Available: https://www.science.org/doi/abs/10.1126/science.aar6404.

[251] E. Ntentos, F. Urdih and U. Zdun, "ML Pipeline Insights Service for Rule-Based Assessment of Training Practices in Reinforcement Learning", in *51th Euromicro Conference Series on Software Engineering and Advanced Applications (SEAA)*, Sep. 2025. DOI: https://doi.org/10.1007/978-3-032-04190-6_10. [Online]. Available: http://eprints.cs.univie.ac.at/8399/.

[252] R. Lowe, Y. Wu, A. Tamar, J. Harb, P. Abbeel and I. Mordatch, "Multi-agent actor-critic for mixed cooperative-competitive environments", *ArXiv*, vol. abs/1706.02275, 2017. [Online]. Available: https://api.semanticscholar.org/CorpusID:26419660.

[253] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad and M. Stal, *Pattern-Oriented Software Architecture*. John Wiley & Sons Ltd, 1996.

[254] DAIR.AI, *Prompt Engineering Guide*, https://www.promptingguide.ai/techniques, 2024.

[255] S. J. Warnett and U. Zdun, *MLOps Pipeline Generation for Reinforcement Learning: A Low-Code Approach Using Large Language Models: Replication Package*, Zenodo, Dec. 2025. DOI: 10.5281/zenodo.18054482. [Online]. Available: https://doi.org/10.5281/zenodo.18054482.

[256] N. Vemuri, N. Thaneeru and V. M. Tatikonda, "AI-Optimized DevOps for Streamlined Cloud CI/CD", *International Journal of Innovative Science and Research Technology*, vol. 9, no. 7, pp. 10–5281, 2024.

[257] B. Zhang, T. Liu, P. Liang, C. Wang, M. Shahin and J. Yu, "Architecture decisions in AI-based systems development: an empirical study", in *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, IEEE, 2023, pp. 616–626.

[258] I. Kumara, R. Arts, D. Di Nucci, W. J. Van Den Heuvel and D. A. Tamburri, "Requirements and reference architecture for MLOps: insights from industry", *Authorea Preprints*, 2023.

[259] P. Sharma and M. S. Kulkarni, "A study on Unlocking the potential of different AI in Continuous Integration and Continuous Delivery (CI/CD)", in *2024 4th International Conference on Innovative Practices in Technology and Management (ICIPTM)*, IEEE, 2024, pp. 1–6.

[260] P. Liang, B. Song, X. Zhan, Z. Chen and J. Yuan, "Automating the Training and Deployment of Models in MLOps by Integrating Systems with Machine Learning", *arXiv preprint arXiv:2405.09819*, 2024.

[261] S. A. A. Naqvi, M. P. Zimmer, P. Drews, K. Lemmer and R. Basole, "The Low-Code Phenomenon: Mapping the Intellectual Structure of Research", in *Hawaii International Conference on System Sciences*, 2024. [Online]. Available: https://api.semanticscholar.org/CorpusID:266757600.

[262] H. El Kamouchi, M. Kissi and O. El Beggar, "Low-code/No-code Development: A systematic literature review", in *2023 14th International Conference on Intelligent Systems: Theories and Applications (SITA)*, 2023, pp. 1–8. DOI: 10.1109/SITA60746.2023.10373712.

[263] M. Hirzel, "Low-Code Programming Models", *Commun. ACM*, vol. 66, no. 10, pp. 76–85, Sep. 2023, ISSN: 0001-0782. DOI: 10.1145/3587691. [Online]. Available: https://doi.org/10.1145/3587691.

[264] J. C. Kirchhof, N. Jansen, B. Rumpe and A. Wortmann, "Navigating the Low-Code Landscape: A Comparison of Development Platforms", in *2023 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, 2023, pp. 854–862. DOI: 10.1109/MODELS-C59198.2023.00135.

[265] O. Bruhin, E. Dickhaut, E. Elshan and M. M. Li, "The Rise of Generative AI in Low Code Development Platforms - An Analysis and Future Directions", in *Hawaii International Conference on System Sciences*, 2024. [Online]. Available: https://api.semanticscholar.org/CorpusID:266757710.

[266] H. Jin, H. Chen, Q. Lu and L. Zhu, "Towards Advancing Code Generation with Large Language Models: A Research Roadmap", *arXiv preprint arXiv:2501.11354*, 2025.

[267] J. Jiang, F. Wang, J. Shen, S. Kim and S. Kim, "A Survey on Large Language Models for Code Generation", *ACM Trans. Softw. Eng. Methodol.*, Jul. 2025, Just Accepted, ISSN: 1049-331X. DOI: 10.1145/3747588. [Online]. Available: https://doi.org/10.1145/3747588.

[268] D. Li and L. Murr, "HumanEval on Latest GPT Models–2024", *arXiv preprint arXiv:2402.14852*, 2024.

[269] M. Chen et al., *Evaluating Large Language Models Trained on Code*, 2021. arXiv: 2107.03374 [cs.LG].

[270] W. Li, D. Zan, B. Guan, A. Yu, X. Chen and Y. Wang, "Improving Natural Language Capability of Code Large Language Model", *arXiv preprint arXiv:2401.14242*, 2024.

[271] Z. Zheng et al., "A Survey of Large Language Models for Code: Evolution, Benchmarking, and Future Trends", *ArXiv*, vol. abs/2311.10372, 2023. [Online]. Available: https://api.semanticscholar.org/CorpusID:265281389.

[272] C. Parnin, G. Soares, R. Pandita, S. Gulwani, J. Rich and A. Z. Henley, "Building Your Own Product Copilot: Challenges, Opportunities, and Needs", *arXiv preprint arXiv:2312.14231*, 2023.

[273] K. Jesse, T. Ahmed, P. T. Devanbu and E. Morgan, "Large Language Models and Simple, Stupid Bugs", in *Proc. 20th International Conference on Mining Software Repositories (MSR)*, IEEE/ACM, May 2023, pp. 563–575. DOI: 10.1109/MSR59073.2023.00082. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/MSR59073.2023.00082.

[274] Z. Xu, S. Jain and M. Kankanhalli, "Hallucination is inevitable: An innate limitation of large language models", *arXiv preprint arXiv:2401.11817*, 2024.

[275] J. White, S. Hays, Q. Fu, J. Spencer-Smith and D. C. Schmidt, "ChatGPT Prompt Patterns for Improving Code Quality, Refactoring, Requirements Elicitation, and Software Design", in *Generative AI for Effective Software Development*, A. Nguyen-Duc, P. Abrahamsson and F. Khomh, Eds. Cham: Springer Nature Switzerland, 2024, pp. 71–108, ISBN: 978-3-031-55642-5. DOI: 10.1007/978-3-031-55642-5_4. [Online]. Available: https://doi.org/10.1007/978-3-031-55642-5_4.

[276] D. Tan and J. Wang, *Engineering Practices for LLM Application Development*, https://martinfowler.com/articles/engineering-practices-llm.html, 2024.

[277] N. M. do Nascimento, P. S. Alencar and D. D. Cowan, "Artificial Intelligence vs. Software Engineers: An Empirical Study on Performance and Efficiency using ChatGPT.", in *CASCON*, 2023, pp. 24–33.

[278] B. Meyer, "AI Does Not Help Programmers", *BLOG@CACM*, 2023.

[279] H. Tian et al., "Is ChatGPT the Ultimate Programming Assistant–How far is it?", *arXiv preprint arXiv:2304.11938*, 2023.

[280] Z. Fan, X. Gao, M. Mirchev, A. Roychoudhury and S. H. Tan, "Automated Repair of Programs from Large Language Models", in *Proc. 45th International Conference on Software Engineering (ICSE)*, Melbourne, Victoria, Australia: IEEE, 2023, pp. 1469–1481, ISBN: 9781665457019. DOI: 10.1109/ICSE48619.2023.00128. [Online]. Available: https://doi.org/10.1109/ICSE48619.2023.00128.

[281] M. Jin et al., "InferFix: End-to-End Program Repair with LLMs", in *Proc. 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2023, 2023, pp. 1646–1656. DOI: 10.1145/3611643.3613892.

[282] L. Belzner, T. Gabor and M. Wirsing, "Large language model assisted software engineering: prospects, challenges, and a case study", in *Proc. International Conference on Bridging the Gap between AI and Reality*, Springer, 2023, pp. 355–374.

[283] J. Yu et al., *An Insight into Security Code Review with LLMs: Capabilities, Obstacles and Influential Factors*, 2024. arXiv: 2401.16310 [cs.SE]. [Online]. Available: https://arxiv.org/abs/2401.16310.

[284] A. Damarapati, "CI/CD Best Practices: Building Reliable Pipelines", *European Journal of Computer Science and Information Technology*, vol. 13, no. 10, pp. 1–10, 2025. DOI: 10.37745/ejcsit.2013/vol13n10110. [Online]. Available: https://doi.org/10.37745/ejcsit.2013/vol13n10110.

[285] M. Fowler, *Patterns of enterprise application architecture*. Addison-Wesley, 2012.

[286] DeepSeek-AI et al., *DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning*, 2025. arXiv: 2501.12948 [cs.CL]. [Online]. Available: https://arxiv.org/abs/2501.12948.

[287] J. Liu, C. S. Xia, Y. Wang and L. Zhang, "Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation", in *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. [Online]. Available: https://openreview.net/forum?id=1qvx610Cu7.

[288] P. Runeson, M. Host, A. Rainer and B. Regnell, *Case study research in software engineering: Guidelines and examples*. John Wiley & Sons, 2012.

[289] S. J. Warnett and U. Zdun, *RLOps Pipeline Development with Low-Code and Large Language Models for Industry 4.0: Replication Package*, Jan. 2026. DOI: 10.5281/zenodo.18320785. [Online]. Available: https://doi.org/10.5281/zenodo.18320785.

[290] M. Towers et al., *Gymnasium: A standard interface for reinforcement learning environments*, 2025. arXiv: 2407.17032 [cs.LG]. [Online]. Available: https://arxiv.org/abs/2407.17032.

[291] T. Übellacker, *AcademiaOS: Automating Grounded Theory Development in Qualitative Research with Large Language Models*, 2024. arXiv: 2403.08844 [cs.HC]. [Online]. Available: https://arxiv.org/abs/2403.08844.

[292] Y. Yue, D. Liu, Y. Lv, J. Hao and P. Cui, "A Practical Guide and Assessment on Using ChatGPT to Conduct Grounded Theory: Tutorial", *J Med Internet Res*, vol. 27, e70122, May 2025, ISSN: 1438-8871. [Online]. Available: https://doi.org/10.2196/70122.

[293] L. Apvrille and B. Sultan, "System Architects Are not Alone Anymore: Automatic System Modeling with AI", in *Proceedings of the 12th International Conference on Model-Based Software and Systems Engineering, MODELSWARD 2024, Rome, Italy, February 21-23, 2024*, F. J. D. Mayo, L. F. Pires and E. Seidewitz, Eds., SCITEPRESS, 2024, pp. 27–38.

[294] R. Keshri, A. Zachariah and M. Boone, *Enhancing Code Consistency in AI Research with Large Language Models and Retrieval-Augmented Generation*, Feb. 2025. DOI: 10.48550/arXiv.2502.00611.

[295] P. Lewis et al., "Retrieval-augmented generation for knowledge-intensive NLP tasks", in *Proceedings of the 34th International Conference on Neural Information Processing Systems*, ser. NIPS '20, Vancouver, BC, Canada: Curran Associates Inc., 2020, ISBN: 9781713829546.

[296] L. Faubel and K. Schmid, "MLOps: A Multiple Case Study in Industry 4.0", in *2024 IEEE 29th International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2024, pp. 01–08. DOI: 10.1109/ETFA61755.2024.10711136.

[297] M. Baldoni, C. Baroglio, V. Ditano, R. Micalizio and S. Tedeschi, "Agents for Industry 4.0: the Case Study of a Production Cell", in *Workshop From Objects to Agents*, 2023. [Online]. Available: https://api.semanticscholar.org/CorpusID:266211460.