

Architecting Reinforcement Learning Pipelines: ADD-Based Insights from an Industry 4.0 Case Study

Evangelos Ntontos and Uwe Zdun

Faculty of Computer Science, Research Group Software Architecture

University of Vienna, Vienna, Austria

firstname.lastname@univie.ac.at

Abstract—Reinforcement Learning (RL) is increasingly used in modern Industry 4.0 systems, enabling intelligent control and adaptation in dynamic production environments. However, the management and operationalization of RL models is less mature than established machine learning operations (MLOps) practices, which are primarily designed for traditional machine learning and deep learning workflows. Unlike these conventional approaches, which rely on static datasets and predefined training pipelines, RL agents learn continuously and interact with evolving environments, adding to the complexity of model management. This work investigates how RL models are managed in practice and how their training pipelines are architected and maintained. We study an industrial RL pipeline through a real production-automation case study, analyzing system artifacts and configurations using a catalog of RL-specific Architectural Design Decisions (ADDs) as the analytical lens. The work contributes a structured catalog of 15 ADDs for RL model management and pipeline implementation and demonstrates how static analysis can assess conformance to these practices. To further support our findings, we present a tool that detects architectural patterns and identifies missing practices, enabling engineers to build more reliable, traceable, and reproducible RL systems.

Index Terms—Reinforcement Learning, Machine Learning, Architectural Design Decisions, Industry 4.0, Cyber-Physical Production Systems, Case Study

I. INTRODUCTION

Industry 4.0 has transformed manufacturing by embedding advanced digital technologies into Cyber-Physical Production Systems (CPPS). This has enabled intelligent automation, real-time monitoring, and flexible operations [1]. Within this context, RL has emerged as an effective method for training agents in realistic simulations to optimize decision-making and processes [2], [3].

Furthermore, MLOps practices are primarily designed to support traditional machine learning and deep learning workflows in production. These practices include automated data processing, model training, deployment, monitoring, and retraining via CI/CD pipelines and model registries [4], [5]. However, applying MLOps principles to RL, commonly referred to as RLOps [6], has novel challenges. Unlike supervised learning, RL agents generate their own training data through environmental interaction, and the target policy may evolve continuously rather than converge to a fixed model. These dynamics complicate versioning, deployment,

and model management of RL policies, making traditional MLOps approaches insufficient for RL use cases.

This paper addresses this gap by focusing on how RL models are managed and how their training pipelines are implemented in an Industry 4.0 setting. We study an industrial RL system to understand how it handles the model lifecycle from configuration and training to deployment, monitoring, and updates. To make the analysis clear and repeatable, we define a set of ADDs for RL model management and pipeline implementation. These ADDs describe the key decision options architects and engineers face, such as how configurations are defined, how models and checkpoints are stored, how retraining is triggered, and how evaluation and reproducibility are managed. By reconstructing these ADDs from system artifacts, we identify which practices follow common MLOps principles, which have been adapted for RL, and which are specific to RL systems used in industrial environments. Unlike MLOps ADDs, several decisions in our catalog are directly driven by RL-specific properties such as continuous interaction, non-stationarity, and safety-critical deployment.

While our prior work [7] synthesizes ADDs for training-strategy in RL architectures, this work shifts the focus to end-to-end RL pipeline and model-management decisions. We study how such decisions are actually realized and adapted in an industrial RL system, and complement this analysis with tool support for assessing architectural conformance.

We aim to answer the following research questions:

- **RQ1:** How are established ADDs for RL model management and pipeline implementation applied in a real-world industrial RL system?
- **RQ2:** To what extent are existing design options of these ADDs adopted and customized in practice to meet the specific requirements of industrial systems?

To answer these questions, we conduct an exploratory case study of an industrial RL-based production system. The study is based on a catalog of 15 ADDs that we define for RL model management and pipeline implementation. We analyze project artifacts and perform static code inspection to see how each ADD and its alternatives appear in practice. This perspective enables us to understand how architectural options influence

RL pipelines and how these choices align with established MLOps principles.

We also build on and extend the *RL Pipeline Insights Service* [8], a previously introduced lightweight static analysis prototype. In this work, we augment the tool with ADD-specific detectors and an improved reporting pipeline so that it can systematically assess architectural conformance in RL pipelines. This extended version allows us to combine manual inspection with automated validation, increasing the consistency and reproducibility of our findings.

Our work makes the following contributions:

- 1) *Empirical Insights on RL Pipeline Design*: We provide an in-depth analysis of how RL model management and pipeline implementation are approached in real-world industrial systems. This includes investigating practical adaptations and challenges in the large-scale application of RL.
- 2) *Catalog of Architectural Design Decisions*: We present a structured set of ADDs and design options that capture key decisions for building and maintaining RL pipelines. The catalog serves as practical guidance for architects and engineers working with RL systems.
- 3) *Static Analysis for Architectural Conformance*: We introduce a method and supporting tool for detecting RL design practices through static analysis. This enables systematic evaluation of RL architectures and supports automation of RLOps compliance checks.

The rest of the paper is organized as follows. Section II summarizes existing work on RLOps, model management, and RL pipeline practices. Section III presents the research method, including data collection, ADD derivation, and the static analysis procedure and case study description. Section IV introduces the catalog of ADDs used to structure the analysis. Section V explains how these ADDs are operationalized for assessing pipeline conformance. Section VI presents the findings for each ADD based on evidence from the industrial system. Section VII discusses how these results answer the research questions and what they imply for RLOps in practice. Section VIII outlines the main threats to validity. Finally, Section IX concludes the paper and describes future research directions.

II. RELATED WORK

In this section, we review studies on RL practices, methods, and MLOps practices, and compare them with our work.

A. RL and MLOps

Sutton and Barto [9] describe reinforcement learning as a paradigm in which an agent learns through repeated interactions with its environment and receives feedback in the form of rewards. This interaction-centric paradigm contrasts with supervised learning, which is based on static datasets. Consequently, the lifecycle of an RL policy includes continuous data generation through experience, online or periodic retraining, and cautious deployment to avoid regressions in live systems [6], [10]. In our approach, we operationalize this lifecycle view

by mapping the industrial case study artifacts to a catalog of ADDs for model management and pipeline implementation, enabling a structured analysis of how these open-ended and safety-critical dynamics are addressed in practice.

In traditional MLOps settings, the machine learning lifecycle is fairly linear and predictable. Kreuzberger et al. [4] describe how data are first collected and versioned, then a model is trained offline, versioned, deployed, and continuously monitored. When the data change or concept drift occurs, the model is retrained [11]. Baylor et al. [12] and Zaharia et al. [13] demonstrate how mature platforms, such as TFX and MLflow, support this process through features like model registries, version tracking, and CI/CD pipelines for automatic retraining and deployment. These tools assume that model artifacts can be stored, compared, and promoted through structured workflows.

However, applying these ideas to reinforcement learning is a challenging task. Li et al. [6] point out that, unlike supervised learning, RL does not rely on a fixed dataset. Instead, the data is created through interaction, and the environment itself can change over time [14]. As a result, the training process is often open-ended, and deployment may require switching policies on the fly while ensuring stability and safety [15]. Compared to these works, our study takes a further step by examining how these lifecycle mechanisms are actually implemented in a real industrial RL system, identifying which MLOps practices transfer, which require adaptation, and where current tooling falls short.

Recent work investigates unique RLOps challenges. Li et al. [6] introduce the term RLOps and articulate differences between ML and RL lifecycles in Open RAN settings, including training stability, deployment safety, and monitoring triggers tied to environment events. This work also demonstrates an RL analytics/operations platform, focusing on high-level lifecycle phases, and argues for additional steps, such as simulation-to-reality transfer and continuous validation, beyond standard MLOps. While their work discusses RLOps at a conceptual and platform level, our study complements it by examining how such lifecycle and pipeline decisions are actually realized in practice in a real industrial RL system.

Other studies show the nascent nature of RLOps. Del Real Torres et al. [16] review deep RL in smart manufacturing and highlight edge deployment and sim-to-real concerns, with emphasis on algorithms rather than operational practices. Kegyes et al. [17] survey RL in Industry 4.0, likewise focusing on methods and applications rather than lifecycle operations. Current domain-oriented reviews, such as Panzer’s study on RL in production planning and control [18], primarily address technical and application-related challenges rather than operationalization.

In contrast, MLOps tools such as TFX demonstrate mature pipelines and CI/CD patterns for traditional ML [12] and model management via registries and serving stacks [13], [19], but do not directly address RL’s continual interaction loop. In contrast to surveys and algorithm-centric research, our ADD-focused approach provides a structured, practice-

oriented method, offering insights that connect abstract RLOps concepts with concrete implementation patterns in an Industry 4.0 environment. Warnett et al. [20] present an industry case examining which MLOps practices transfer to RL and what needs adaptation. In contrast to this work’s focus on existing widely used MLOps practices in an RL context, we focus on model management and pipeline implementation practices that are specific for RL.

B. Model Management and Pipeline Implementation

In MLOps, model versioning is a core practice. Zaharia et al. [13] and Kreuzberger et al. [4] demonstrate that every trained model should be stored as a versioned artifact, along with metadata such as the data used, the code commit, hyperparameters, and evaluation metrics. This enables comparison of different runs, A/B testing, and rollback when needed. Baylor et al. [12] and Olston et al. [19] further describe how the full lifecycle training, deployment, monitoring, and updates are automated through CI/CD pipelines and supported by model registries. While these works describe model versioning and lifecycle automation in general ML systems, our study examines how these ideas appear in a real industrial RL system and how design choices differ when versioning, checkpointing, and deployment must account for continuous interaction with a changing environment.

For RL, versioning becomes more complex. Liang et al. [10] and Baylor et al. [12] note that RL training produces many intermediate policies over episodes or timesteps, and choosing which version to deploy is not straightforward. In contrast to these works, our study maps these RL-specific challenges to explicit ADDs, enabling us to precisely observe where the industrial system adheres to these practices and where gaps or deviations occur in practice.

Furthermore, several studies focus on the importance of well-structured ML pipelines to ensure reproducibility and continuous delivery. Steidl et al. [21] describe a four-stage framework for continuous ML development, data handling, model learning, software development, and system operations, highlighting how ML pipelines differ from classical DevOps flows in their triggers, branching, and orchestration strategies. Similarly, other work stresses modularization, automated metric capture, and reproducible execution as key enablers of reliable ML pipelines [4]. These implementation-level concerns map directly to our ADD set for RL pipeline implementation and management, where configuration, logging, checkpointing, and reproducibility are central design choices.

III. RESEARCH METHOD

This section explains the research methodology illustrated in Figure 1. Presenting the method first clarifies how the ADDs were derived, how the data were collected, and how the analysis proceeded. The data used in and produced as part of this study have been made available online for reproducibility [22].

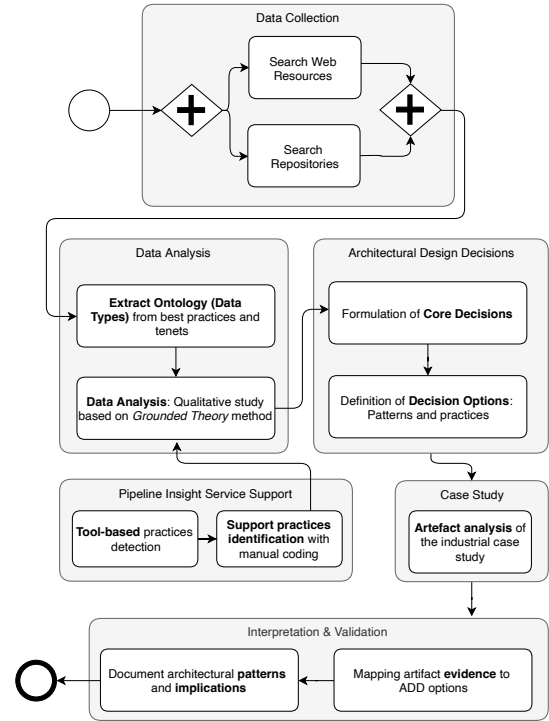


Fig. 1. Overview of the research method followed in this study.

A. Data Collection

We adopt a case study methodology following established guidelines for empirical software engineering by Runeson and Höst [23]. Our chosen unit of analysis is the system artifacts involved in RL/ML management. More specifically, we gathered:

- Source code:** the main RL orchestration scripts and related modules that implement training routines, model saving/loading, and deployment logic;
- CI/CD pipeline definitions:** configuration files and scripts that define automated processes (for training, evaluation, deployment) triggered by events;
- Architecture documentation:** high-level design diagrams and internal documentation describing the system’s architecture and data flows;
- Version control history:** commit logs and comments in the source repository that give insight into how the code evolved (especially changes related to model updates or pipeline tweaks);
- Informal discussions:** clarifications from the development team obtained through our ongoing collaboration (to understand ambiguous code or design decisions, as interviews were not formally conducted).

B. Industrial Case Study

The object of our case study is an advanced RL-based software system developed by an industrial partner for automating manufacturing processes in an Industry 4.0 setting. The case is a proprietary CPPS that integrates one or more RL agents into

a production line. The RL agents learn to perform tasks such as the physical assembly of components and adapt to changes in the manufacturing environment. The system leverages MLOps/ROps practices to manage the training and deployment of these agents. The case embodies a real-world ROps pipeline within a factory automation solution. This makes it a good example of how model management and pipeline implementation are handled in practice. The development team comprises domain experts in manufacturing, RL researchers, and software engineers. The project has been under active development, incorporating continuous improvements, and has resulted in the production of a variety of model versions and pipeline adjustments over time.

Figure 2 illustrates one RL pipeline of the industrial case study. The pipeline follows a modular design and automates key stages of the RL lifecycle, reflecting several of the identified ADDs.

It begins with the *environment setup* phase, where the case simulation environment and agent interfaces are instantiated and configured (ADD 14: Environment Compatibility Management). This is followed by *policy creation and algorithm configuration*, where the RL algorithm, here based on Q-learning, is defined together with its hyperparameters (ADD 7: Hyperparameter Exploration). During the *training phase*, the policy interacts with the simulated environment to optimize its behavior; intermediate results are periodically stored as *model checkpoints* (ADD 4: Checkpointing, ADD 5: Training Initialization Strategy). These checkpoints allow training to resume after interruptions and support later model reuse or warm-start initialization.

After training, the pipeline performs *model analysis and visualization*, enabling inspection of training curves and performance metrics (ADD 1: Logging). Subsequently, the trained policy is *loaded and evaluated* under fixed seeds and controlled conditions to assess its stability and generalization (ADD 6: Evaluation and Generalization Testing Protocol).

All steps are automated and executed locally within a unified RL training environment, demonstrating partial automation of the RL lifecycle while maintaining transparency and reproducibility.

C. Data Analysis

Our analysis builds on the model-based qualitative approach introduced in [24], which combines Grounded Theory (GT) [25], [26] with pattern mining [27], [28] to derive architectural decision models from practitioner-oriented “grey literature” (e.g., documentation, engineering reports, blog posts, and technical guidelines). Following this approach, we synthesized a catalog of ADDs prior to the case study through a structured grey-literature mining process, based on in our previous work [7]. Practitioner sources were identified through keyword-based searches related to reinforcement learning pipelines and model lifecycle management (e.g., “model versioning RL”, “RL experiment management”, “RL training pipelines”). We used major search engines (e.g., Google, Bing, DuckDuckGo) and practitioner-oriented

technology portals (e.g., InfoQ, DZone) to identify relevant sources. Candidate sources were screened using lightweight inclusion and exclusion criteria. We included practitioner-oriented materials describing concrete engineering practices, architectural solutions, or operational experiences with RL or ML pipelines. We excluded purely academic papers, marketing content without technical substance, and duplicate discussions of the same practice. The screening and coding process was performed by two researchers and iteratively refined through discussion to mitigate individual bias. Relevant sources were analyzed using a constant-comparison process following GT principles. Concepts related to architectural decisions and implementation practices were coded and iteratively grouped into candidate ADDs and design options. The search and coding process continued until theoretical saturation was reached, i.e., when additional sources did not yield new decision categories or options. The industrial case study then played a complementary role by instantiating, validating, and refining this catalog through systematic artifact inspection. Source code and pipeline configurations were analyzed to corroborate literature-derived ADD options and to identify system-specific refinements, which were used to extend or clarify existing options rather than introduce new ADDs. Manual inspection constituted the primary analysis method; the extended RL Pipeline Insights Service was used to systematically support and triangulate findings by identifying ADD-related indicators that might otherwise be overlooked, rather than to validate manual results.

Each new observation either confirmed an existing ADD option or revealed a new system-specific option. This iterative process concluded when no additional options emerged.

D. Tool Support for Analysis

To complement manual coding and reduce researcher bias, we employed an extended version of the *RL Pipeline Insights Service*. The original prototype was introduced in prior work [8]. In this study, we adapted and expanded it with ADD-specific detectors so that it can systematically identify architectural patterns related to the decisions defined in Section IV.

The tool processes source code and configuration artifacts using a set of targeted, rule-based detectors. Each detector corresponds to one or more ADD indicators. The tool architecture consists of:

- **Central Orchestrator:** Coordinates execution and manages the flow of detector modules.
- **Detectors:** Heuristic and rule-based components that search for ADD-specific implementation patterns or anti-patterns.
- **Code Analyzer:** Parses files and applies detectors to relevant code regions.
- **Reporting Component:** Produces structured JSON outputs indicating the presence, absence, or partial implementation of ADD-related indicators.

The tool does not replace manual inspection, nor does it evaluate architectural quality. Instead, it operationalizes predefined ADD indicators as observable implementation patterns

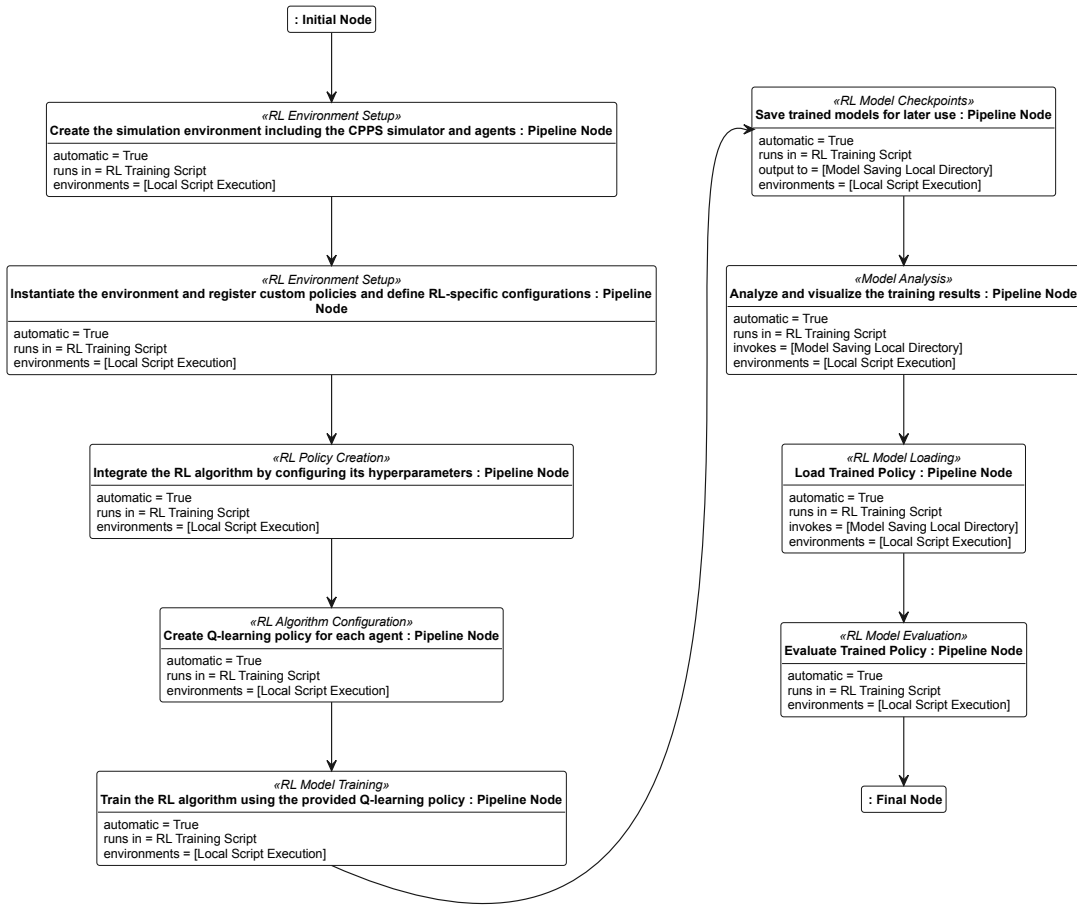


Fig. 2. An example RL pipeline diagram of the case study.

and systematically detects their presence or absence within the codebase. In this sense, it functions as an analysis instrument that supports consistency and traceability in the coding process, rather than as an independent evaluation mechanism. Its outputs are used to triangulate manual findings and reduce the risk of overlooking relevant implementation artifacts. The structured JSON reports integrate into the conformance assessment procedure described in Section V, enabling reproducible and transparent analysis across the RL pipeline artifacts.

The tool does not replace manual inspection; rather, it provides a systematic mechanism to verify and cross-check observations. Its outputs integrate into the assessment procedure described in Section V, supporting consistent, reproducible evaluation across the RL codebase.

E. Ethical Considerations

Confidentiality and proper handling of sensitive industrial information were maintained throughout the study. No human subjects were involved beyond routine collaboration with the industrial partner. The partner provided informed consent and reviewed the findings to ensure no proprietary details were disclosed. All artifacts were handled under existing confidentiality agreements. No inducements were offered, and no conflicts of interest are declared.

IV. ARCHITECTURAL DESIGN DECISIONS FOR RL MODEL MANAGEMENT AND PIPELINE IMPLEMENTATION

This section presents the catalog of ADDs derived using the research method described in Section III. These ADDs define recurring architectural choices in the RL pipeline design. Further details on the ADDs and their decision options are provided in the replication package [22].

A. Architectural Design Decisions

To guide our analysis, we formulated a catalog of 15 ADDs that describe the key recurring choices involved in managing RL models and implementing their pipelines. Each ADD represents an important design question, such as how retraining is triggered, how configurations and checkpoints are stored, how hyperparameters are tuned, or how evaluation and reproducibility are ensured. For every ADD, we identified a set of design alternatives from both MLOps/RL Ops research and the industrial case study.

Tables I and II summarize the ADDs and their decision options. These decisions form the foundation for our case study analysis, helping to identify recurring architectural patterns and implementation practices in RL systems.

TABLE I
ARCHITECTURAL DESIGN DECISIONS FOR RL MODEL MANAGEMENT AND PIPELINE IMPLEMENTATION (ADD 1–8)

ADD	Design options
ADD 1: Logging	Op1: Unstructured console/file logging Op2: Structured logging (JSON, CSV) with unified schema Op3: Centralized log aggregation and visualization Op4: No logging
ADD 2: Version Identifier Semantics	Op1: Sequential numeric identifier Op2: Semantic versioning with stability or policy-breaking flags Op3: Timestamp-based identifier Op4: Composite identifier Op5: Metadata-augmented version (includes environment, algorithm, or seed info) Op6: No versioning mechanism
ADD 3: Model Registry	Op1: Centralized registry service storing model artifacts, lineage, and metadata Op2: Integrated artifact store with indexing and retrieval API Op3: Version-control-backed storage with model pointers and metadata tracking Op4: Local/ad-hoc storage (no central registry)
ADD 4: Checkpointing	Op1: Manual model checkpointing at user-defined intervals Op2: Automatic periodic checkpointing during training Op3: Event-driven checkpointing (based on reward improvement thresholds) Op4: Distributed checkpoint management with metadata and retention policy Op5: No checkpoint mechanism
ADD 5: Training Initialization Strategy	Op1: Training from scratch Op2: Warm start from the latest production model Op3: Warm start from the best historical checkpoint Op4: Progressive or ensemble-based warm start
ADD 6: Evaluation and Generalization Testing Protocol	Op1: Single-environment evaluation only Op2: Multi-environment validation Op3: Baseline comparison against static or heuristic policies Op4: Automated benchmark evaluation with statistical summaries
ADD 7: Hyperparameter Exploration	Op1: Single training run without hyperparameter optimization Op2: Grid or random search over hyperparameters Op3: Adaptive search (e.g., Bayesian optimization, ASHA, Hyperband) Op4: Multi-algorithm exploration (e.g., PPO, SAC, DDPG variants)
ADD 8: Configuration Management Strategy	Op1: Static configuration files (e.g., YAML, JSON) committed with code Op2: Centralized configuration service (dynamic parameters, remote updates) Op3: Hierarchical configuration Op4: Runtime parameter injection via CLI or environment variables

V. PATTERN-BASED PIPELINE CONFORMANCE

This section explains how the ADD catalog is operationalized when analyzing the industrial RL system.

A. Operationalizing ADDs for Code and Artifact Inspection

For each ADD, we defined a set of observable indicators that can be identified directly in system artifacts. These indicators map the abstract design options from Tables I and II to concrete implementation patterns. Examples include:

- **Logging (ADD 1):** presence of structured log messages, consistent schema fields, or timestamped event traces.
- **Version identifiers (ADD 2):** naming conventions for model files, timestamp-based folder structures, or semantic version tags.
- **Checkpointing (ADD 4):** calls to save model state, checkpoint retention logic, or event-driven checkpoint triggers.
- **Initialization strategy (ADD 5):** loading of previous model artifacts or invocation of warm-start logic.

These indicators make each ADD verifiable in practice, allowing the abstract catalog to be applied systematically.

B. Assessment Procedure

The conformance assessment combines:

- 1) *Manual inspection of artifacts* (configuration files, training scripts, orchestration code), where indicators are identified through targeted searches and code reading;
- 2) *Static analysis outputs* from the tool described in Section III, which flags occurrences of ADD-related patterns and potential gaps.

For each ADD option, we record whether it is:

- **Implemented** (direct evidence is present),
- **Partially implemented** (evidence is incomplete or only implicit),
- **Not implemented** (no evidence found).

This produces the conformance overview presented later in the findings.

Conformance Analysis. The goal of this analysis is not to introduce new contributions, but to describe the technical bridge between the ADD catalog and the case study findings. By defining concrete indicators and a systematic assessment procedure, the ADDs become operational tools for architectural evaluation. This step ensures transparency and reproducibility in how the results in Section VI were obtained.

VI. ANALYSIS AND FINDINGS

Through our analysis of the industrial case study, we examined how model management and pipeline implementation

TABLE II
ARCHITECTURAL DESIGN DECISIONS FOR RL MODEL MANAGEMENT AND PIPELINE IMPLEMENTATION (ADD 9–15)

ADD	Design options
ADD 9: Release and Rollback Policy	Op1: Manual approval Op2: Automated threshold-based promotion Op3: Multi-criteria promotion Op4: Rollback mechanism integrated into CI/CD pipelines
ADD 10: Evaluation Protocol	Op1: Offline evaluation using simulation or replay data Op2: Online shadow evaluation Op3: Controlled A/B or canary evaluation with safety guardrails Op4: Composite metric set (e.g., reward, domain KPIs, safety scores) Op5: Statistical testing with confidence thresholds
ADD 11: Modular Pipeline Architecture	Op1: Monolithic RL script Op2: Layered modularization (environment, agent, training, evaluation) Op3: Component-based architecture Op4: Pipeline orchestration via workflow manager
ADD 12: Policy Artifact Format	Op1: Framework-native weights (e.g., PyTorch, TensorFlow) Op2: Intermediate portable format (e.g., ONNX, TorchScript) Op3: Containerized model image embedding runtime dependencies Op4: Custom binary format with checksum and digital signature
ADD 13: Artifact and Metric Data Formats	Op1: Ad-hoc storage of metrics Op2: Structured data serialization Op3: Unified experiment schema Op4: Registry-based artifact metadata tracking
ADD 14: Environment Compatibility Management	Op1: No compatibility verification Op2: Manual environment compatibility checks before deployment Op3: Automated compatibility validation
ADD 15: Retraining Triggers	Op1: Time-/schedule-based retraining Op2: Performance-degradation triggers Op3: Environment-change triggers Op4: Manual/on-demand retraining trigger Op5: Hybrid trigger policy

practices are realized in practice. To structure our findings, we organize them around key ADDs reflecting central aspects of model management. Each finding is supported by architectural evidence observed in the analyzed artifacts.

- 1) Logging and Traceability
- 2) Model Versioning and Registry Infrastructure
- 3) Checkpointing, Recovery, and Training Initialization
- 4) Evaluation, Selection, and Deployment
- 5) Configuration, Orchestration, and Retraining

A. Logging and Traceability (ADD 1)

The system implements a consistent structured logging format across the main components. A unified formatter standardizes timestamps, severity levels, and process identifiers, which yields uniform traceability across training, evaluation, and orchestration workflows. This structured approach supports observability and reproducibility, and aligns with the *structured logging with unified schema* practice. However, logs remain local to the execution environment and are not aggregated or visualized through a centralized monitoring backend. As a result, real-time monitoring, anomaly detection, and cross-run analytics remain limited.

B. Model Versioning and Registry Infrastructure (ADD 2–3)

Model versioning is present but remains largely implicit. Version identifiers embed metadata such as a CRC of the configuration, component name, algorithm class, and episode index. This provides uniqueness and supports manual traceability, even though no semantic or human-readable versioning

scheme is used. Model storage follows naming conventions within local directories rather than a dedicated registry or metadata management service. Artifacts are not indexed, queried, or linked through automated lineage, and rollback requires manual file selection. Overall, model versioning is metadata-rich but developer-oriented, while registry functionality is still limited to structured local storage without governance or automation.

C. Checkpointing, Recovery, and Training Initialization (ADD 4–5)

Checkpointing is partially realized through periodic or on-demand exports of learned policies. Some components implement configurable checkpoint frequencies, and policies can be restored when explicitly requested. Nonetheless, checkpointing is not uniformly coordinated across the system, and no global retention, promotion, or cleanup logic is integrated. Training typically starts from scratch unless the user manually enables restoration. Although warm starts are technically supported, they are not the default operational mode. This approach simplifies experimentation but limits continuity, reproducibility, and recovery from interruptions.

D. Evaluation, Selection, and Deployment (ADD 6–10)

Evaluation is conducted in simulated environments using domain-specific KPIs and includes comparison against a baseline configuration. This provides meaningful feedback for scheduling and control tasks. However, evaluations are limited to single-environment scenarios, without multi-environment

stress testing or statistical significance analysis. Long-term aggregation is supported through JSON summaries but remains offline and manually interpreted. Model selection and deployment are manual processes. Although policies can be exported and invoked by downstream components, no automated scoring, canary testing, health checks, or rollback workflows are integrated. As a result, evaluation and deployment are effective for experimentation but do not realize a full continuous evaluation or safe-deployment pipeline.

E. Configuration, Orchestration, and Retraining (ADD 8, 11–15)

The pipeline implementation follows a clear modular structure, with separate components for policy definition, environment interaction, optimization, and evaluation. This modularization aligns with the *layered pipeline architecture* pattern, supporting maintainability, testing, and reusability across various use cases. However, configuration and orchestration mechanisms remain lightweight. Parameters are set through internal environment variables or static configuration blocks, lacking centralized schema management or hierarchical overrides. No automated retraining or event-based orchestration was detected. Retraining decisions are manually initiated by operators, and hyperparameter exploration is performed through static definitions rather than adaptive search. Compatibility management across environments and policies is also manual, relying on developer awareness rather than automated validation scripts. These characteristics suggest that while the pipeline achieves architectural clarity, it has not yet adopted full automation or dynamic control.

F. ADD Coverage Across the Pipeline

To provide a consolidated overview of how all architectural design decisions manifest in the industrial system, Table III summarizes the applicability of the 15 ADDs related to RL model management and pipeline implementation. It indicates whether each decision option was applied (✓), not applied (✗), or only partially realized (◦). This cross-cutting summary complements the ADD-specific findings presented above and offers a holistic view of the system’s architectural maturity. Table IV provides additional interpretation of evidence and architectural implications.

VII. DISCUSSION

The findings from the industrial RL case study offer concrete answers to our research questions. They illustrate how ADDs for RL model management and pipeline implementation are manifested in practice and which design options are applied, adapted, or omitted.

A. RQ1: Reflection of RL ADDs in an Industrial System

Our results show that the case indicates that all fifteen ADDs are applicable within this industrial RL context. Practices such as structured logging (ADD 1), modular and layered pipeline architecture (ADD 11), and configuration-driven component separation (ADD 8) are clearly implemented and provide a

stable architectural foundation. These ADDs support maintainability, transparency, and consistent experimentation.

Other ADDs, such as model registry integration (ADD 3), coordinated checkpointing (ADD 4), warm-start strategies (ADD 5), and automated retraining orchestration (ADD 15), are only partially realized. Rather than employing end-to-end automation, the system relies on structured naming conventions and locally stored artifacts to maintain continuity. These mechanisms offer basic traceability but fall short of the guarantees provided by registry or orchestration frameworks.

B. RQ2: Design Options Applied and Adapted in Practice

When examining the decision options implemented, we observed that the implementation prioritizes simplicity and controlled human oversight over full automation. Applied options include structured logging, timestamp-based version identifiers, modular pipeline orchestration, and manually triggered retraining. Automated features, such as centralized model registries, rollback policies, or event-driven retraining, were not yet deployed. Based on discussions with industrial partners, these capabilities were considered desirable but not yet operationalized due to safety and organizational readiness. At the same time, several options were consciously adapted to suit RL’s dynamic and safety-critical nature: Retraining is manually initiated only after human validation of stable environmental conditions has been completed. Compatibility checks between the model and the environment are embedded into the deployment steps instead of running as automated scripts. Configuration parameters are injected at runtime to allow flexible experimentation within controlled limits.

These adaptations suggest that reliability and controlled deployment are prioritized over full automation. The system aligns with many best-practice principles but implements them conservatively to ensure operational safety. These adaptations are specific to RL systems operating in physical environments, where automated retraining or rollback may pose safety risks not present in data-centric ML pipelines.

C. Architectural Implications

The analyzed system illustrates a transition from ad-hoc RL experimentation toward a more structured, yet still deliberately supervised, RLOps pipeline. Its modular architecture, structured logging, and configuration control establish a solid foundation for traceability and maintainability, while the absence or partial realization of practices such as model registry integration, automated checkpoint governance, and coordinated retraining positions the system at an intermediate level of RLOps maturity. In this setting, stability and interpretability are achieved through intentional human oversight rather than full automation. The findings suggest that further progress toward continuous model evolution will require integrating model registries, standardizing artifact schemas, and introducing carefully controlled automation for version tracking and performance-driven retraining. More broadly, the results highlight that effective RLOps in industrial RL systems is not purely a technical challenge, but also an organizational

TABLE III
 COVERAGE OF DESIGN OPTIONS (OP1–OP5) FOR EACH ARCHITECTURAL DESIGN DECISION (ADD1–ADD15) IN THE CASE STUDY, BASED ON THE ANALYZED CODE ARTIFACTS. ✓ = APPLIED; ◦ = PARTIALLY APPLIED OR IMPLICIT; × = NOT APPLIED.

ADD	Op1	Op2	Op3	Op4	Op5	Op6
ADD1: Logging	×	✓	×	×	—	—
ADD2: Version Identifier Semantics	×	×	×	✓	✓	×
ADD3: Model Registry	×	×	×	✓	—	—
ADD4: Checkpointing	◦	✓	×	◦	×	—
ADD5: Training Initialization Strategy	✓	◦	×	×	—	—
ADD6: Evaluation & Generalization Protocol	✓	×	✓	×	—	—
ADD7: Hyperparameter Exploration	✓	✓	×	×	—	—
ADD8: Configuration Management Strategy	✓	×	×	✓	—	—
ADD9: Release & Rollback Policy	×	×	×	×	—	—
ADD10: Evaluation Protocol	✓	×	×	✓	×	—
ADD11: Modular Pipeline Architecture	×	✓	×	×	—	—
ADD12: Policy Artifact Format	✓	×	×	×	—	—
ADD13: Artifact & Metric Data Formats	✓	✓	◦	×	—	—
ADD14: Environment Compatibility Mgmt	✓	×	×	—	—	—
ADD15: Retraining Triggers	×	×	×	✓	×	—

one, involving conscious trade-offs between automation and human control. For researchers, this underscores the value of pattern-based RLOps guidance that links architectural design decisions with everyday operational practices, helping bridge the gap between RL system design and deployment in practice.

D. Lessons for Practitioners

The findings translate into several concrete lessons for designing or maintaining RL pipelines:

- **Apply structured logging and configuration traceability.** These practices provide immediate gains in observability and reproducibility, even without full registry or orchestration support.
- **Adopt incremental versioning early.** Even simple timestamp-based identifiers and consistent naming conventions can provide a foundation for future registry migration.
- **Strengthen checkpointing and warm-start capabilities.** Coordinated checkpointing and consistent restore mechanisms can significantly reduce training costs and improve resilience.
- **Modularize before automating.** Clear separation of environment, agent, and evaluation components is essential before introducing orchestration or CI/CD workflows.
- **Align governance with automation.** Human-in-the-loop workflows remain crucial in early RLOps maturity stages, but should gradually be complemented by automated validation gates and drift monitoring.

VIII. THREATS TO VALIDITY

In this section, We discuss the threats to validity based on the threat types by Wohlin et al. [29].

a) *Construct validity:* Our identification of ADDs and patterns relied on content analysis of architectural artifacts, source code, and configuration files. While we rely on evidence from multiple artifacts to reduce bias, there is a risk

of misinterpretation or overlooking implicit decisions that are not well-documented in the artifacts. Informal discussions with system engineers clarified some uncertainties; however, the lack of structured interviews means that certain design rationales may still be unclear. Although our static analysis followed widely accepted best practices, applying them to the case study artifacts required some subjective judgment, which may have influenced how specific ADDs were coded.

As with any grey-literature-based synthesis, the ADD catalog is not claimed to be exhaustive; rather, it reflects dominant practices observed across practitioner sources and is intended as a practical analytical lens. While this may limit completeness, our goal is not to enumerate all possible RL pipeline decisions, but to capture recurring and actionable practices relevant to industrial contexts.

b) *Internal validity:* Our observations of architectural practices are based on system artifacts and limited operational insights rather than controlled experiments. Although the available evidence supports our interpretations, some implementation choices may have been influenced by organizational constraints rather than deliberate architectural design. For instance, the adoption of staged rollouts followed an early production failure rather than a planned design decision. Such situational factors can shape how practices develop, making it difficult to draw clear causal conclusions.

c) *External validity:* This study examines a single industrial RL system used in an Industry 4.0 production environment. While the case study represents a cutting-edge RL deployment, its findings may not fully generalize to other fields. The identified ADDs and patterns are likely relevant across domains, but their importance and implementation details may vary. Future research, such as multiple case studies or broader surveys, would help assess the extent to which these findings are applicable.

TABLE IV
FINDINGS ON RL MODEL MANAGEMENT AND PIPELINE IMPLEMENTATION IN THE INDUSTRIAL CASE STUDY

ADD	Evidence in the System	Implications
ADD 1: Logging	Consistent structured logging format is applied across all main components.	Enables traceability and debugging, though the lack of log aggregation reduces operational visibility.
ADD 2: Version Identifier Semantics	Models and checkpoints encode metadata such as CRC, component name, algorithm, and episode, but lack human-readable semantic versioning.	Provides uniqueness and metadata richness but limits interpretability and version meaning for developers or operators.
ADD 3: Model Registry	Models are stored in local folders using naming conventions rather than a registry service.	Allows manual management but restricts discoverability, lineage tracking, and systematic rollback.
ADD 4: Checkpointing	Checkpoints are saved periodically or on demand, with partial support for configurable frequency but no global retention or cleanup policy.	Supports basic recovery and warm-starting, but inconsistent patterns reduce reproducibility and maintainability.
ADD 5: Training Initialization Strategy	Training usually starts from scratch but can optionally restore a previous checkpoint when configured.	Allows reuse of prior models when explicitly chosen but is not uniformly leveraged across runs.
ADD 6: Evaluation and Generalization	Agents are evaluated in simulated environments with domain KPIs and baseline comparison, but no multi-environment or statistical tests.	Provides meaningful environment-specific evaluation but limited robustness and generalization insights.
ADD 7: Hyperparameter Exploration	Standard runs use fixed hyperparameters, while a dedicated tuning script performs grid search over parameter combinations.	Enables systematic tuning when invoked but lacks adaptive or automated hyperparameter optimization.
ADD 8: Configuration Management	Experiments rely on configuration files with CLI/environment overrides for flexibility.	Balances reproducibility and adaptability, but distributed overrides may lead to hidden configuration drift.
ADD 9: Release and Rollback Policy	No automated promotion or rollback logic is implemented; deployment decisions occur outside the code.	Ensures human oversight but provides no automated safeguards or rollback triggers.
ADD 10: Evaluation Protocol	Evaluation produces structured JSON results with domain KPIs; all evaluation is offline in simulation.	Supports post-run analysis but lacks online testing or statistically grounded decision thresholds.
ADD 11: Modular Pipeline Architecture	The pipeline shows a modular structure with separate components for environments, optimization, training, evaluation, and orchestration.	Enhances maintainability and reuse, though some utilities remain more monolithic.
ADD 12: Policy Artifact Format	Policies are stored in framework-native formats such as checkpoints and serialized weight files.	Ensures compatibility with current frameworks but reduces portability across tools or platforms.
ADD 13: Artifact and Metric Data Formats	Metrics and artifacts are stored mostly as JSON with some ad-hoc elements.	Allows structured analysis within the project but lacks centralized schema governance or tracking tools.
ADD 14: Environment Compatibility Management	Compatibility relies on correctly implemented wrappers and mappings; no automated compatibility checks exist.	Functionality depends on developer discipline, increasing potential for hidden mismatches.
ADD 15: Retraining Triggers	All retraining and tuning activities must be initiated manually.	Keeps full human control but prevents automated adaptation or drift-aware retraining.

IX. CONCLUSION AND FUTURE WORK

This study examined how RL model management and pipeline implementation are realized in an industrial setting using an ADD perspective. By analyzing artifacts from a real Industry 4.0 production system, we identified how RL-specific practices align with, extend, or diverge from traditional MLOps principles.

Our analysis reveals that the system has a clear and well-structured architecture, with several RLOps practices already in place. Ideas such as a modular pipeline, consistent logging, and well-organized configuration files provide the system with a solid foundation for maintaining stability and ease of traceability. However, some important practices, such as using a proper model registry, automating checkpointing, or triggering retraining based on system events, are still missing or only partially implemented. As a result, much of the RL lifecycle is still partially manual rather than being fully automated.

These findings suggest that industrial RL systems are currently in a transitional stage, transitioning from ad-hoc experimentation toward structured yet not yet fully automated RLOps pipelines. The primary challenge lies in striking a bal-

ance between automation and operational safety and reliability in cyber-physical environments.

For practitioners, the results show the value of architectural thinking when building RL systems. Even partial implementation of practices such as structured logging, versioning, and checkpointing can significantly enhance aspects like reproducibility and lifecycle visibility.

Future work. Future work includes studying additional industrial RL systems, improving our analysis to better detect semantic patterns, and examining related areas such as environment versioning alongside model versioning. Strengthening tool support is also important; integrating RLOps checks directly into CI/CD pipelines could help teams enforce these practices in development. Multi-agent RL and federated RL, bring their own lifecycle challenges, including synchronization and distributed rollback, which will require additional practices and architectural solutions.

Acknowledgments This research was funded in whole or in part by the FFG (Austrian Research Promotion Agency) project MODIS (no. FO999895431) and project BEAM, (no. FO999933314).

REFERENCES

- [1] S. J. Oks, M. Jalowski, M. Lechner, S. Mirschberger, M. Merklein, B. Vogel-Heuser, and K. M. Mösllein, "Cyber-physical systems in the context of industry 4.0: A review, categorization and outlook," *Information Systems Frontiers*, vol. 26, no. 5, pp. 1731–1772, 2024.
- [2] W. Zhang, F. Zhao, Y. Li, C. Du, X. Feng, and X. Mei, "A novel collaborative agent reinforcement learning framework based on an attention mechanism and disjunctive graph embedding for flexible job shop scheduling problem," *Journal of Manufacturing Systems*, vol. 74, pp. 329–345, 2024.
- [3] Y. Lin, H. Zhao, and H. Ding, "Real-time path correction of industrial robots in machining of large-scale components based on model and data hybrid drive," *Robotics and Computer-Integrated Manufacturing*, vol. 79, p. 102447, 2023.
- [4] D. Kreuzberger, N. Kühl, and S. Hirschl, "Machine learning operations (mlops): Overview," *Definition, and Architecture. arXiv*, vol. 20222205, 2022.
- [5] 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)*. IEEE, 2021, pp. 25–28.
- [6] P. Li, J. Thomas, X. Wang, A. Khalil, A. Ahmad, R. Inacio, S. Kapoor, A. Parekh, A. Doufexi, A. Shojaeifard *et al.*, "Rlops: Development lifecycle of reinforcement learning aided open ran," *IEEE Access*, vol. 10, pp. 113 808–113 826, 2022.
- [7] E. Ntentos, S. J. Warnett, and U. Zdun, "Supporting architectural decision making on training strategies in reinforcement learning architectures," in *21st IEEE International Conference on Software Architecture ICSEA*, June 2024.
- [8] 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)*, September 2025.
- [9] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, 2nd ed. MIT Press, 2018. [Online]. Available: <http://incompleteideas.net/book/the-book-2nd.html>
- [10] E. Liang, R. Liaw, R. Nishihara, P. Moritz, R. Fox, K. Goldberg, J. E. Gonzalez, M. I. Jordan, and I. Stoica, "Rllib: Abstractions for distributed reinforcement learning," in *Proceedings of the 35th International Conference on Machine Learning (ICML 2018)*, 2018, pp. 3053–3062. [Online]. Available: <https://proceedings.mlr.press/v80/liang18b.html>
- [11] J. Gama, I. Žliobaitė, A. Bifet, M. Pechenizkiy, and A. Bouchachia, "A survey on concept drift adaptation," *ACM computing surveys (CSUR)*, vol. 46, no. 4, pp. 1–37, 2014.
- [12] D. Baylor, E. Breck, H.-T. Cheng, N. Fiedel, C. Y. Foo, Z. Haque, S. Haykal, M. Ispir, V. Jain, L. Koc *et al.*, "Tfx: A tensorflow-based production-scale machine learning platform," in *Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining*, 2017, pp. 1387–1395.
- [13] M. Zaharia, A. Chen, A. Davidson, A. Ghodsi, S. A. Hong, A. Konwinski, S. Murching, T. Nykodym, P. Ogilvie, M. Parkhe, F. Xie, and C. Zumar, "Accelerating the machine learning lifecycle with mlflow," *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, vol. 41, no. 4, pp. 39–45, 2018. [Online]. Available: https://people.eecs.berkeley.edu/~matei/papers/2018/ieec_mlflow.pdf
- [14] Z. Fang and U. Zdun, "Detecting environment drift in reinforcement learning using a gaussian process," in *2024 IEEE 36th International Conference on Tools with Artificial Intelligence (ICTAI)*. IEEE, 2024, pp. 992–999.
- [15] J. Garcia and F. Fernández, "A comprehensive survey on safe reinforcement learning," *Journal of Machine Learning Research*, vol. 16, no. 1, pp. 1437–1480, 2015.
- [16] A. del Real Torres, D. S. Andreiana, A. Ojeda Roldan, A. Hernandez 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, p. 12377, 2022.
- [17] T. Kegyes, Z. Süle, and J. Abonyi, "The applicability of reinforcement learning methods in the development of industry 4.0 applications," *Complexity*, vol. 2021, no. 1, p. 7179374, 2021.
- [18] M. Panzer and B. Bender, "Deep reinforcement learning in production systems: A systematic literature review," *International Journal of Production Research*, vol. 60, no. 13, pp. 4316–4341, 2022.
- [19] C. Olston, N. Fiedel, K. Gorovoy, J. Harmsen, L. Lao, F. Li, V. Rajashekhar, S. Ramesh, and J. Soyke, "Tensorflow-serving: Flexible, high-performance ml serving," *arXiv preprint arXiv:1712.06139*, 2017. [Online]. Available: <https://arxiv.org/abs/1712.06139>
- [20] S. J. Warnett and U. Zdun, "Bridging the gap between mlops and rlops: An industry 4.0 case study on architectural design decisions," in *2025 IEEE International Conference on Software Architecture (ICSA)*, 2025, to appear; preprint available. [Online]. Available: https://eprints.cs.univie.ac.at/8343/1/Bridging_the_Gap_Between_MLOps_and_RLOps_eprints.pdf
- [21] 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. 111615, 2023.
- [22] "Replication package." [Online]. Available: <https://doi.org/10.5281/zenodo.18619901>
- [23] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," *Empirical Software Engineering*, vol. 14, no. 2, pp. 131–164, 2009. [Online]. Available: <https://doi.org/10.1007/s10664-008-9102-8>
- [24] U. Zdun, M. Stocker, O. Zimmermann, C. Pautasso, and D. Lübke, "Guiding architectural decision making on quality aspects in microservice apis," in *International Conference on Service-Oriented Computing*. Springer, 2018, pp. 73–89.
- [25] B. G. Glaser and A. L. Strauss, *The Discovery of Grounded Theory: Strategies for Qualitative Research*. de Gruyter, 1967.
- [26] K. Charmaz, *Constructing grounded theory*. Sage, 2014.
- [27] J. Coplien, *Software Patterns: Management Briefings*. SIGS, New York, 1996.
- [28] 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*, 2015, pp. 9:1–9:16.
- [29] C. Wohlin, P. Runeson, M. Hoest, M. C. Ohlsson, B. Regnell, and A. Wesslen, *Experimentation in Software Engineering*. Springer, 2012.