

Agentic AI Architecture for Evaluating and Improving Reinforcement Learning Pipelines

EVANGELOS N TENTOS, UWE ZDUN, Faculty of Computer Science, Research Group Software Architecture, University of Vienna, Austria

Reinforcement Learning (RL) systems are gaining traction in application areas such as robotic systems, self-driving vehicles, and automated industrial operations. In these safety-critical domains, lifecycle management practices – including model versioning, registry integration, automated evaluation, retraining triggers, and staged deployments – are just as important as algorithmic performance in ensuring reliability and reproducibility. However, these practices are often scattered across source code, configuration files, CI/CD workflows, and deployment manifests, making it difficult to automate detection and correction. This paper introduces an Agentic AI architecture that utilizes multiple specialized LLM-based agents to evaluate RL pipelines against a curated catalog of 15 architectural best practices. The agents are specialized (e.g., file selection, detection, fix suggestion, code generation), and an orchestrated pipeline with automated validation gates combines practice detection and LLM reasoning to identify both explicit and implicit practice gaps and suggest, or where appropriate, even apply fixes. We validate the approach in a large-scale industrial cyber-physical production case study and in five open-source RL repositories with diverse architectures, demonstrating the scalable detection of missing practices and the generation of actionable fixes. The system achieved an overall detection F_1 -score of 0.80 and a mean generation-quality score of 2.40, indicating that it reliably identifies missing practices and produces functionally valid code artifacts.

Additional Key Words and Phrases: Reinforcement Learning, Best Practices, Machine Learning, Architecture Agentic AI, Case Studies, LLMs


ACM Reference Format:

Evangelos Ntentos, Uwe Zdun. 2026. Agentic AI Architecture for Evaluating and Improving Reinforcement Learning Pipelines. In *2026 IEEE/ACM 5th International Conference on AI Engineering - Software Engineering for AI (CAIN '26)*, April 12–13, 2026, Rio de Janeiro, Brazil. ACM, New York, NY, USA, 18 pages. <https://doi.org/10.1145/3793653.3793759>

1 INTRODUCTION

Reinforcement Learning (RL) systems are increasingly deployed in safety-critical domains such as robotics, autonomous vehicles, and industrial control. Their long-term reliability depends not only on algorithmic choices (e.g., policy gradients, Q-learning) and algorithmic performance but also on applying rigorous engineering practices throughout the RL model lifecycle. Modern RL toolkits, such as Stable-Baselines3 [24], Pytorch [23], Tensorflow [1] RLLib [19], and CleanRL [12], lower the barriers to experimentation but offer only little guidance on lifecycle management. As a result, practices such as centralized model versioning, automated evaluation, staged deployments, and retraining triggers are often implemented inconsistently or left to ad-hoc scripts, undermining reproducibility, safety, and maintainability in real-world deployments.

Author's address: Evangelos Ntentos, Uwe Zdun, firstname.lastname@univie.ac.at, Faculty of Computer Science, Research Group Software Architecture, University of Vienna, Vienna, Austria.

Please use nonacm option or ACM Engage class to enable CC licenses 
This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2026 Copyright held by the owner/author(s).
Manuscript submitted to ACM

Manuscript submitted to ACM

Evaluating whether an RL pipeline conforms to best practices is challenging. Conventional static analysis tools or heuristic-based detectors [22] can reliably identify obvious recurring code patterns (e.g., checkpoint calls, fixed hyper-parameters) but struggle when practices are scattered across multiple files, embedded in infrastructure configurations (e.g., Helm charts, Kubernetes manifests), or implemented via CI/CD and workflow orchestrators. These limitations are particularly evident in large-scale deployments, where essential practices such as registry usage, version pinning, or rollback logic often occur as indirect dependencies or API interactions rather than explicit code.

To address these challenges, we propose an *Agentic AI* architecture that coordinates multiple specialized Large Language Model (LLM)-based agents to detect gaps in model versioning and lifecycle management, recommend corrections, and, where appropriate, either apply automated corrections or submit proposed changes for human review. We use an agentic architecture because the evaluation and correction of lifecycle practices span diverse artifacts and reasoning tasks. Dividing the work into role-specific agents improves accuracy, traceability, and controllability while enabling targeted validation and human oversight. Each agent focuses on a specific role (e.g., file selection, detection, fix suggestion, code generation), and the system orchestrates them through a collaborative pipeline with automated validation gates. This design enables our architecture to combine the precision of practice analysis with the reasoning and generalization capabilities of LLMs.

This paper addresses the following research questions:

- **RQ1:** To what extent can a coordinated agentic system based on LLM recognize whether RL pipelines comply with best practices for model versioning and lifecycle management, including cases where these practices are distributed across code, infrastructure, and CI/CD workflows?
- **RQ2:** How effectively can such a system suggest and apply corrections (patches) to comply with best practices?

Our contributions are (i) an agentic evaluation and correction architecture for RL lifecycle management that orchestrates specialized LLM-based agents and validation gates to balance automation with human oversight; (ii) a catalog of *15 architectural practices* covering important aspects of RL model lifecycle including centralized model versioning, automated evaluation, staged deployments, and retraining triggers; and (iii) an empirical evaluation demonstrating practical scalability and usefulness for improving RL reproducibility and safety.

We evaluate our approach in two settings: (i) a large-scale industrial case study in production automation, where multiple RL pipelines manage distributed processes, and (ii) five open-source RL systems representing diverse architectures and coding practices. Together, these settings demonstrate both the scalability and adaptability of the Agentic AI architecture.

The remainder of the paper is structured as follows: Section 2 reviews RL lifecycle best practices. Section 3 discusses related work in RL code analysis and AI-assisted detection. Section 4 presents the agentic AI detection architecture, multi-LLM workflow, and rule set. Section 5 describes the industrial and open-source evaluation scenarios. Section 6 reports quantitative and qualitative results. Section 7 interprets the findings, and Section 8 outlines threats to validity. Finally, Section 9 summarizes our contributions and future work.

2 BACKGROUND ON REINFORCEMENT LEARNING PRACTICES

This section provides background on the RL practices for RL Model Versioning and Lifecycle Management that we focus on in this work.

Managing the lifecycle of RL models effectively requires a set of consistent and well-structured practices [18, 26]. A major part of this process is a centralized model registry, which acts as the single source of truth for all trained models

[10]. Every model stored in this registry should be accompanied by detailed metadata, including its version number, the training environment, hyperparameters, and performance metrics. This not only ensures transparency but also enables traceability when models are reused, compared, or rolled back. To maintain system security and ensure a clear separation between components, all interactions with the registry, training pipelines, deployment orchestrators, and inference services should be handled exclusively through well-defined remote APIs [6].

For deploying models, infrastructure-as-code practices should be standard. Using Kubernetes manifests or Helm charts ensures that deployments are scalable, reproducible, and easy to roll back when needed. Automated rollback mechanisms should be in place so that if performance drops or deployment errors occur, the system can revert to a stable version without manual intervention [29].

Before any model is promoted to production, it should go through a standardized evaluation step within the pipeline [4, 9]. Predefined performance metrics help ensure that only the best candidates move forward. Once a model passes this evaluation, its promotion to production should be automated to eliminate human bias and speed up release cycles [5].

Retraining is another critical part of the lifecycle [33]. Operational indicators, such as changes in the environment or performance degradation, should be monitored automatically, with configurable thresholds stored in formats like YAML or JSON. When a retraining event is triggered, it should be logged in a structured way, capturing the reason for the trigger, the relevant metrics, and the final outcome [4, 5].

When a model successfully completes training in a simulation environment, it should be automatically registered in the model registry along with its metadata [31]. Deployments must always reference specific version numbers to prevent the accidental use of untested models. Additionally, separate stages, such as development, staging, and production, should be maintained, with clear distinctions between model versions in each. Techniques like canary releases or A/B testing can help ensure that new models are introduced safely.

3 RELATED WORK

In this section, we discuss the studies that focus on RL practices and rule-based detection methods, and we compare them with our work.

Many studies have focused on and addressed different aspects of RL methodologies. For instance, Lee et al. [16] investigate the evolution of RL algorithms, highlighting the transition from single-agent systems to multi-agent configurations that rely on distributed optimization. Canese et al. [2] analyze multi-agent algorithms, emphasizing crucial elements such as scalability, non-stationarity, and observability, all of which are vital in multi-agent RL contexts.

Eimer et al. [7] provide an in-depth analysis of hyperparameter tuning approaches, covering techniques like grid search, random search, and Bayesian optimization. Li et al. [17] propose the Hyperband method for hyperparameter optimization, which efficiently reduces computational demands while identifying optimal parameters. Our agentic AI approach complements these by focusing not on new algorithms but on detecting, suggesting, and correcting practices in RL pipelines through multi-agent analysis and validation.

Chen et al. [3] explore the use of checkpoints in training processes, presenting methods to improve model performance by periodically saving and validating models. Similarly, Eisenman et al. [8] explore checkpointing systems in training recommendation models, outlining how checkpoints can be utilized to boost training efficiency. Our agentic AI approach extends this by automating checkpoint detection and validation directly in source code, using LLM-based reasoning to suggest and apply corrections.

Hernandez-Leal et al. [11] present a comprehensive review of multi-agent deep RL, addressing the challenges of non-stationarity, scalability, and agent cooperation. Our work addresses these challenges by focusing on model versioning and lifecycle management practices and validating them using an LLM-based and multi-agent-based approach.

Zhang et al. [32] provide an extensive overview of MARL algorithms, focusing on cooperative and competitive dynamics, underscoring the need for strong training practices. Our work complements these insights by implementing a multi-agent LLM-based framework and validating RL practices used in real-world systems.

There are also several studies address the architectural aspects of model lifecycle and version management in ML systems. Miao et al. [20] introduce *ModelHub*, a unified framework for managing deep learning artifacts through versioning and provenance tracking. Le et al. [15] propose *VeML*, an end-to-end lifecycle system that treats data, models, and workflows as versioned entities, enabling reproducibility and drift detection. These works focus on supervised ML, while our work extends lifecycle management to reinforcement learning through automated multi-agent LLM reasoning.

4 APPROACH

This section describes the research methods followed in this study and our *Agentic AI* architecture, which uses multiple task-specific LLM agents to evaluate RL pipelines against a curated set of best practices for model versioning and lifecycle management. Our approach enables the systematic and scalable evaluation and correction of RL systems through the autonomous execution of the detection and correction workflow. This includes finding relevant code, configurations, and documentation, evaluating compliance across files, recommending improvements, and generating concrete code or configuration corrections. The data used in and produced as part of this study have been made available online for reproducibility¹.

4.1 Research Methodology

Figure 1 illustrates the overall research process followed in this study. We began by reviewing a range of sources on RL best practices, including academic literature [7, 14, 25, 27, 30], online resources, and open-source repositories. The collected material was then analyzed qualitatively using Grounded Theory methods [28], applying open and axial coding to identify recurring patterns and derive key practices. To ensure coverage of common model lifecycle concerns, we also drew on established pattern catalogs such as those by Lakshmanan [14]. This review resulted to the creation of a catalog of RL best practices (Section 2), covering aspects such as centralized model registries, consistent versioning, automated evaluation and promotion, retraining triggers, structured logging, and staged deployments.

We have operationalized these practices in an agentic architecture, where specialized LLM-based agents collaborate to detect, interpret, and resolve discrepancies in code, configuration, and CI/CD artifacts. The multi-agent pipeline performs cross-file and cross-format reasoning to evaluate complex, polyglot codebases and infrastructure manifests that are challenging to assess using static or heuristic-only methods. The agents generate compliance assessments with supporting evidence, prioritized suggestions, and, where possible, concrete code or configuration patches that can be automatically validated or escalated for human review. We then evaluated our approach across six case studies, both industrial and open-source, to assess the detection performance and the quality of the generated artifacts.

This approach contains four main steps:

¹The tool is published in a long-term repository Zenodo: <https://doi.org/10.5281/zenodo.17422753>

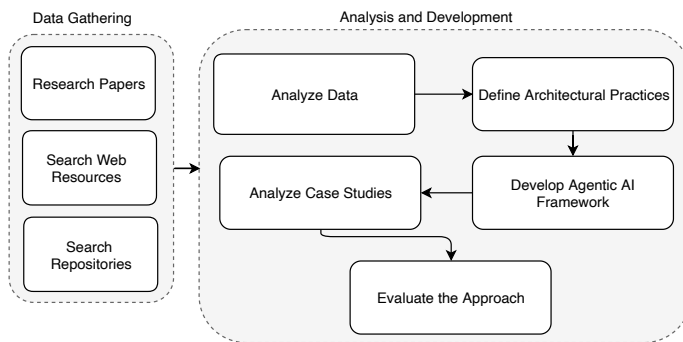


Fig. 1. Overview of the research method followed in this study.

Step 1. Identification of Best Practices: Through an extensive literature review and qualitative analysis, we identified a comprehensive set of RL best practices that enhance reliability, traceability, and reproducibility in model operations. Table 1 summarizes these practices, focusing on model versioning, lifecycle management, and deployment.

Step 2. Agent Role Design: We designed four specialized LLM-based agents to evaluate compliance with best practices. The *File Selection Agent* locates relevant source or configuration files, the *Detection Agent* determines whether the practice is implemented, the *Fix Suggestion Agent* proposes improvements for missing elements, and the *Code Generation Agent* produces and escalates code updates for review (see Section 4.2).

Step 3. Implementation of the Agentic AI Architecture: The framework follows the *Model Context Protocol (MCP)* [13], enabling standardized context exchange and message passing between agents. Specialized agents communicate via a shared memory and orchestrator, reusing intermediate reasoning traces and validation results. The modular MCP-based design allows seamless integration of new agents or best-practice definitions while maintaining transparency and human oversight.

The framework orchestrates the specialized agents using a message-passing mechanism. Each agent is guided by role-specific prompts, ensuring consistent and interpretable outputs. The modular design supports the addition of new agents or best practice definitions without disrupting the workflow, enabling continuous evolution of detection capabilities.

Step 4. Application to Case Studies and Validation: We applied the framework to two evaluation settings:

- (1) **A large-scale industrial case study** involving RL-driven Cyber-Physical Systems for production automation, where multiple pipelines were monitored for adherence to best practices such as version-pinned deployments, automated retraining triggers, and structured retraining logs.
- (2) **Five open-source RL systems** covering a diverse range of architectures and coding styles.

4.2 Agentic AI Architecture

Figure 2 presents a high-level view of the Agentic AI architecture. The framework follows a deterministic, validator-gated workflow for RL codebases, structured into four stages: *select*, *detect*, *suggest*, and *generate code*. Execution is supervised by a *Collaboration Protocol Agent (CPA)*, which controls orchestration decisions without modifying code, and a lightweight *Automated Validation Agent (AVA)*, which performs structural checks without LLM-based reinterpretation.

Table 1. Model Versioning and Lifecycle Management Practices

ID	Description
PR1	Record structured metadata including configurations, random seeds, environment versions, and metrics. This ensures full experiment reproducibility and auditability.
PR2	Capture structured runtime logs with traces, correlation IDs, and service metrics. Such observability aids debugging and performance tracking.
PR3	Store trained models with immutable version identifiers and metadata such as commit ID and environment details. This preserves reproducibility and traceability.
PR4	Always load models using pinned version numbers. Deterministic loading prevents inconsistencies between training and deployment.
PR5	Register all models in a centralized registry accessible via APIs. The registry manages metadata, lineage, and model accessibility.
PR6	Automate model selection based on quantitative metrics and safety checks. Multi-metric evaluation or human review prevents biased promotions.
PR7	Use standardized evaluation pipelines with predefined datasets and metrics. This enables consistent comparison of model performance across runs.
PR8	Perform automated validation of deployment artifacts for schema, dependency, and performance compatibility. It prevents runtime errors in production.
PR9	Implement rollback mechanisms triggered by failed evaluations or unstable deployments. This ensures rapid recovery to a known good state.
PR10	Maintain versioned API contracts defining schema and endpoint stability. This safeguards clients from unexpected interface changes.
PR11	Design retraining pipelines with configurable triggers based on drift, performance, or schedule. Include staging validation before production rollout.
PR12	Conduct safe rollouts through canary or A/B testing strategies. Define acceptance thresholds and rollback plans to limit risk.
PR13	Log retraining events with details on triggers, metrics, and final outcomes. This creates a traceable audit trail for lifecycle monitoring.
PR14	Use clear stage separation (development, staging, production) with versioned manifests. It enables controlled progression and rollback.
PR15	Test models in small-scale real-world or digital-twin environments before full deployment. This mitigates operational risks early.

More specificity, the **File Selection Agent** identifies candidate files using configurable include/exclude rules. The **Detection Agent** evaluates these files against targeted practices via structured LLM prompts. When gaps are identified, the **Suggestion Agent** proposes corrective actions, which are realized by the **Code Generation Agent** through code or configuration updates. At each stage, the AVA enforces validation gates (e.g., file type, evidence presence, syntax, linter status), while the CPA advances the workflow by issuing a single decision per step: continue, retry, succeed, or fail.

4.2.1 *Stage Criteria of Automated Validation Agent.* Automated Validation Agent enforces simple, objective rules:

Select: All files must be source code; file content must contain substantive code (not only comments or print/log lines).

Empty selection is also a fail.

Detect: For every practice marked *supported*, at least one credible code excerpt must be present

Suggest: Every file flagged as needing change must have ≥ 1 actionable suggestion.

Generate Code: All generated or applied patches must pass language-specific syntax checks and be free of critical linter errors. If patches were expected but none could be successfully applied, this phase fails, and AVA indicates errors per file with diagnostic messages and logs.

4.2.2 CPA Decision Protocol. The *Collaboration Protocol Agent* acts as the central orchestrator of the Agentic AI workflow. It sequences the execution flow (Select → Detect → Suggest → Generate Code), inserts validation gates after each stage, and manages conditional retries or loop-backs when validation confidence is low. The CPA coordinates all interactions through a structured message bus and a shared context store, maintaining a run ledger that records agent inputs, outputs, and validation results for full traceability and reproducibility.

This coordination mechanism is implemented following the MCP [13], which defines standardized interfaces for context exchange, memory persistence, and tool invocation among autonomous agents. By adhering to MCP principles, the CPA can orchestrate specialized agents in a composable and interpretable manner, enabling transparent message routing, cross-agent memory consistency, and safe escalation to human oversight when necessary. This MCP-based decision protocol ensures that the multi-agent workflow remains modular and easily extensible with new practices or validation components.

4.2.3 Validation Gates & Retries. After each major stage, the CPA triggers the AVA. If checks pass, the pipeline proceeds. If they fail, the CPA performs limited, evidence-driven retries (e.g., refine file filters, adjust a patch).

4.2.4 Shared Context. All agents read and write to a *Shared Context Store* that contains project metadata, repository structure, detector findings, suggested patches, and validation outputs.

4.2.5 Reporting & Outputs. A *Compliance Report Generator* compiles results into a structured JSON format, including all insights for each stage (e.g., a list of selected files, detection evidence, etc.).

4.2.6 Modularity & Extensibility. The design is modular, allowing agents to be swapped or scaled independently without affecting the overall pipeline. Adding a new best practice requires only its detection logic and suggestion/implementation templates; the CPA and validation gates remain unchanged. Different environments are supported by tuning the file selection.

4.3 Best Practice and Agentic AI Detectors

Best Practice Detection. Table 1 outlines the RL model versioning and lifecycle management best practices defined in this study. Each best practice is evaluated as either *supported* or *not supported* within a given RL pipeline, along with code evidence that supports its evaluation. A practice is considered supported if its core intent is realized in the system’s codebase, configurations, or deployment infrastructure. Conversely, it is marked as not supported if there is either no evidence of the practice or explicit implementation contradicts the intent.

For example, in the case of the practice “*Centralized model registry storing model metadata and accessible via defined APIs.*” (PR5), it would be marked as *supported* if the code or deployment scripts clearly interact with a registry service (e.g., MLflow, ModelDB) and persist model metadata. In contrast, the practice “*Automated model selection*” (PR6) would be marked as *not supported* if model promotion decisions are made manually without automated evaluation.

Agentic AI Detectors & Validators. The Agentic AI architecture utilizes specialized LLM-based agents to assess compliance with best practices and, when necessary, propose and implement fixes. Compared to purely static or

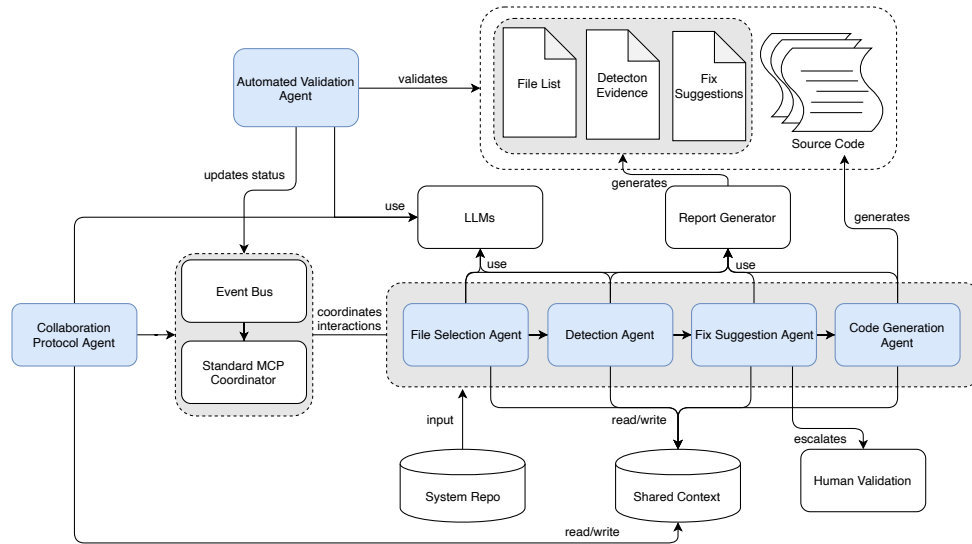


Fig. 2. Workflow of the Agentic AI architecture.

heuristic detectors, this multi-agent setup supports deeper contextual reasoning, cross-file correlation, and recognition of polyglot implementations.

Evaluation Workflow. The basic workflow is a pipeline with validation gates and controlled retries: *File Selection* → *Detection* → *Suggestion* → *Code Generation*. After each major step, the CPA triggers the AVA. If AVA passes, the pipeline continues. If AVA fails or is uncertain, the CPA attempts limited, evidence-driven retries (e.g., refine selection filters, adjust a patch). All intermediate results are stored in a *Shared Context Store* and an audit trail is created, allowing later agents to reason with full history.

For example, the practice “*Structured metadata logging*” (PR1), the *File Selection Agent* identifies logging modules and retraining scripts; the *Detection Agent* checks for JSON logs with triggers, timestamps, and metrics; if logs are unstructured, the *Suggestion Agent* proposes changes; the *Code Generation* produces a minimal patch that adds structured logging to the training loop. The AVA validates each stage (e.g., proper file selected, detection includes evidence, etc.). If AVA fails, CPA triggers a retry at the corresponding stage.

5 CASE STUDIES

In this section, we describe the case studies used to evaluate our approach and test the performance of the framework. We studied one large-scale industry case study and five open-source RL-based systems. The case studies are summarized in Table 2.

5.1 Industrial Case Study

To validate the effectiveness of our Agentic AI architecture, we applied it to a comprehensive industrial case study in the domain of production automation. The system under study was developed by a global provider of automation solutions and is based on a sophisticated RL platform that supports both single-agent and multi-agent setups. This industrial

Table 2. Overview of Python-based case study systems and their functionalities

<i>ID</i>	<i>Description</i>
ICS	The Industrial Case Study is an RL platform for production automation that provides single- and multi-agent simulations, configurable training environments, MLOps-integrated deployment pipelines, an infrastructure for continuous monitoring and retraining, and direct integration into production control loops to optimize resource allocation, scheduling, and process control. It enables thorough testing and optimization of AI policies in diverse environments (See Section 5.1 for details).
CS1	This system, HHMARL 2D, simulate hierarchical multi-agent air combat scenarios, where heterogeneous aircraft agents perform fight-or-escape maneuvers coordinated by a high-level commander’s policy. https://github.com/IDSIA/hhmarl_2D
CS2	This system, NFVdeep, applies deep reinforcement learning to dynamically orchestrate service function chains in network function virtualization environments. https://github.com/CN-UPB/NFVdeep
CS3	This system, IMIL (Infosys Model Inference Library), offers a unified, high-performance framework for loading and deploying machine learning models across diverse platforms and formats. https://github.com/Infosys/Infosys-Model-Inference-Library
CS4	This system applies distributed Proximal Policy Optimization (PPO) to manage multi-agent traffic light control in a SUMO-based urban grid, where each intersection acts as an independent reinforcement learning agent. https://github.com/maxbrenner-ai/Multi-Agent-Distributed-PPO-Traffic-light-control
CS5	This system, JAT (Jack of All Trades), is a multi-purpose transformer agent capable of handling diverse reinforcement learning and vision-language tasks using a unified architecture. https://github.com/huggingface/jat

platform is used in a factory to standardize and improve production processes, relying on RL-driven decision-making to optimize resource allocation, scheduling, and process control.

The platform integrates several key components:

- **Agent Management Modules:** These modules enable the simulation of both individual and multi-agent behaviors, supporting the study of collaborative and competitive dynamics in production workflows.
- **Configurable Training Environments:** The system provides flexible training environments where task parameters, constraints, and dynamics can be adjusted to match real-world production requirements.
- **Deployment Pipelines with MLOps Integration:** Automated CI/CD pipelines manage the transition of RL models from development to deployment, including versioning, evaluation, and rollback strategies.
- **Monitoring and Retraining Infrastructure:** Performance metrics are continuously tracked, and retraining pipelines can be triggered automatically based on performance degradation or environment changes.
- **Production Automation Application:** The RL framework is embedded into real-world automation scenarios, where models are deployed in control loops that directly influence manufacturing processes.

We integrated the *Agentic AI architecture* into the industrial MLOps pipelines to monitor compliance with the best practices. Instead of relying on static heuristics, our system orchestrates specialized LLM-based agents.

This setup enables the real-time and continuous evaluation of the RL pipelines against best practices. By embedding the framework into the MLOps workflows, we are able to:

- Detect missing or inconsistent implementations of practices.
- Automatically generate compliance reports for ML engineers and architects.
- Suggest and apply fixes that ensured consistent adherence to practices across heterogeneous pipelines.

Through this industrial case study, we demonstrate the practical value of the Agentic AI approach in a production-critical setting. The multi-agent LLM framework proved capable of detecting compliance issues that heuristic detectors would overlook, while providing actionable suggestions and automated fixes. This validation highlights the applicability of our approach to real-world MLOps environments where RL models are continuously deployed, monitored, and retrained.

5.2 Open-source Case Studies

The open-source case studies in Table 2 (CS1-CS5) cover various RL algorithms, including Proximal Policy Optimization (PPO), Deep Q-Network (DQN), and Multi-Agent Deep Deterministic Policy Gradient (MADDPG). These systems vary widely in scope, from multi-agent platforms for different types of experiments to advanced PPO setups with complex neural networks. While 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 self-play for competitive games or tasks within defined environments. Some systems are designed to be flexible, allowing for dynamic interactions in 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.

6 VALIDATION

In this section, we validate our Agentic AI architecture by demonstrating its applicability and effectiveness through a series of scenario-based evaluations. Instead of validating isolated practices, we focus on realistic conditions where multiple best practices interact across the RL model lifecycle. Each scenario group combines two or more practices and tests whether the multi-agent framework can (i) detect missing practices, (ii) suggest improvements, and (iii) generate concrete fixes.

6.1 Validation Setup

The validation setup aims to rigorously assess both the accuracy *detection* and the *quality generation* of the proposed Agentic AI architecture for the industrial and open-source case studies. To enable reproducible and objective evaluation, two complementary ground truths were established: one for practice detection accuracy and another for code generation quality.

Ground Truth. The ground truth for detection was created through manual inspection of the source code, configuration, and deployment files of each case study. We annotated the presence or absence of each defined practice (see Table 1) following clear operational definitions to ensure annotation consistency. For generation quality, we evaluated each file generated or modified by the framework using the four-level scale introduced in Section 6.3, scoring each artifact according to completeness, correctness, and compliance with the intended practice.

Agentic AI Execution. The full Agentic AI pipeline was executed for each case study. At each stage, the CPA coordinated agent outputs and enforced validation gates using the AVA. The resulting detection reports and generated artifacts were compared against the manually curated ground truths. This process enabled a unified evaluation of how accurately the framework identifies missing practices and how effectively it generates valid and complete code files.

Model Configuration. The LLM-based agents were instantiated using *gpt-4o-2025-01-01-preview*. This specific model version was selected for consistency and replicability across all evaluation phases. While our framework is designed to be *LLM-agnostic*, supporting interchangeable backends such as *GPT-3.5-Turbo*, *GPT-4.1*, *GPT-5*, and non-GPT models like *LLaMA 3.1* and *Qwen 2.5* we found in prior experiments [22?] that *GPT-4o* offered the most balanced performance. Specifically, it achieved superior detection accuracy and reasoning consistency across diverse RL code samples, while maintaining a favorable computational cost. Although newer models (e.g., *GPT-4.1*, *GPT-5*) may offer incremental reasoning improvements, their higher latency and cost make them less practical for large-scale multi-agent orchestration, where efficiency and reproducibility are prioritized.

Evaluation Metrics. For the detection stage, we used standard classification metrics to quantify the accuracy of practice identification:

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}, \quad \text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

$$\text{F1-score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

True Positives (TP) represent correctly detected practices, False Positives (FP) correspond to incorrect detections, and False Negatives (FN) denote missed practices. True Negatives (TN) are excluded since only positive practice evidence was annotated. For the generation stage, we aggregated mean and distributional statistics of the ground truth of evaluating generated artifacts across agents, practices, and case studies. Both provide a dual evaluation of the framework’s ability to both detect and improve compliance through autonomous reasoning and code generation.

6.2 Validation Process

For each case study, we compared the results of the Agentic AI architecture with the established ground truth. The process followed these steps:

- (1) **Evaluation Setup:** Instantiate the evaluation within the industrial pipeline or an open-source project.
- (2) **Framework Execution:** Run the framework, allowing agents to select files, detect practices, propose fixes and generate solutions.
- (3) **Comparison:** Compare the agent outputs with the manually validated ground truth.
- (4) **Outcome Assessment:** Record whether the framework correctly detected compliance/non-compliance and whether the generated fixes were valid.

6.3 Ground Truth Design and Evaluation Metrics

To ensure a rigorous evaluation of the Agentic AI architecture, we establish two complementary ground truths: one for practice detection accuracy and another for code generation quality. Together, these ground truths provide a comprehensive view of the framework’s performance across its two main dimensions: identifying missing implementation of practices and generating corresponding corrective artifacts.

Ground Truth for Detection. The first ground truth captures the correctness of the *practice detection phase*, where the framework identifies whether a given system implements specific RL practices. A manually curated reference dataset was created to annotate each project file or component with binary labels indicating the presence or absence of a given practice. The framework’s detection reports are compared to this ground truth, and performance is quantified using

precision, recall, and F_1 -score metrics. This evaluation assesses how effectively the system identifies practices at the detection level prior to any remediation or generation.

Ground Truth for Generation Quality. The second ground truth measures the *quality and completeness* of the code artifacts produced by the framework after detection. In Table 3 we introduce a four-level ordinal metric termed the *Ground Truth Quality Score (GQS)*, which reflects the degree to which each generated or modified file satisfies the intended practice.

Table 3. Ground Truth Quality Score for Evaluating Generated Artifacts

Score Label	Description
1 <i>No Valid Output</i>	No file was generated, or the output is syntactically invalid, irrelevant, or empty.
2 <i>Minimal Output</i>	File(s) were generated, but only minor or placeholder content was produced. Structural changes only; core logic missing.
3 <i>Functional but Incomplete</i>	The file(s) contain meaningful and relevant logic aligned with the intended practice; however, additional refinement or validation is required for full compliance.
4 <i>Complete & Valid</i>	Generated code is syntactically correct, functionally complete, and aligned with the expected practice. Passes basic validation (syntax, linter, or policy checks).

For the detection, annotations serve as binary ground truth labels used to compute precision, recall, and F_1 -score. For the GQS, we assess each generated artifact according to the scale in Table 3. Aggregated GQS scores and F_1 -metrics across agents, practices, and case studies provide a dual perspective: (i) the accuracy of the framework’s detection capabilities, and (ii) the quality and completeness of its generated artifacts.

Scale Interpretation. The GQS follows a four-point Likert-type scale that reflects ordered levels of artifact quality rather than exact numerical distances. This means the step from 1 to 2 is not necessarily equivalent to the step from 3 to 4. Accordingly, GQS values are treated as ordinal categories for descriptive comparison, not as interval data for statistical inference. This approach aligns with standard guidance regarding Likert-type scales and the distinction between ordinal and interval measurement levels [21].

6.4 Results

6.4.1 Detection Results. Table 4 summarizes the results of the detection phase from the Agentic AI architecture using *GPT-4o* as the underlying LLM for detection reasoning. The evaluation was conducted against the manually established ground truth (see Section 6), and the reported values correspond to the framework’s precision, recall, and F_1 -score across all case studies. The framework achieved a strong performance, with an F_1 -score of 0.80, indicating that the multi-agent pipeline reliably detects RL practices across diverse repositories.

Overall precision (0.82) slightly exceeded recall (0.78), suggesting a conservative detection pattern in which the framework flags practices only when clear structural or semantic evidence is found. This aligns with the design of the Detection Agent, which prioritizes evidence-based reasoning over heuristics. More specifically, performance was particularly high in CS3 and CS5 ($F_1 = 1.0$), where practices were explicitly implemented, while moderate accuracy was observed in the industrial case (ICS, $F_1 = 0.73$) and in CS1 and CS4 ($F_1 = 0.80$). Lower detection performance occurred in

CS2 ($F_1 = 0.67$), where practices were either absent or weakly implemented. These variations highlight how the pipeline’s precision is influenced by project structure and visibility of practices.

The results confirm that *GPT-4o*, when used through the Agentic AI pipeline, provides a reliable foundation for the subsequent suggestion and code-generation phases.

Table 4. Detection Performance of the Agentic AI Architecture Using GPT-4o

Case Study	Precision	Recall	F1 Score
ICS	0.80	0.67	0.73
CS1	0.80	0.80	0.80
CS2	0.67	0.67	0.67
CS3	1.00	1.00	1.00
CS4	0.80	0.80	0.80
CS5	1.00	1.00	1.00
Total	0.82	0.78	0.80

6.4.2 File Generation Results. After the detection phase, we evaluated the *quality and completeness* of the code artifacts produced by the framework. Each generated or modified file across the six case studies (ICS–CS5) was manually inspected and assigned a GQS ranging from 1 to 4, reflecting how well each artifact satisfied its intended practice. A score of 1 corresponds to invalid or irrelevant output (e.g., broken or empty files), while a score of 4 represents complete and valid implementations that passed all policy and configuration checks. This evaluation provides a reproducible and human-interpretable metric for assessing the quality of the system’s generated output.

Table 5 summarizes the number of generated or modified files and the corresponding mean GQS values per case study. These averages are aligned with the detailed practice-level scores shown in Table 6, providing a consistent overview of the generation quality across all case studies. Higher mean scores indicate that most generated files reached functional and syntactic validity ($GQS \geq 3$), while lower scores reflect partial implementations or missing contextual information that limited code synthesis.

Although Table 5 reports mean GQS values for ease of comparison, these averages should be interpreted with caution. The GQS is an ordinal scale, meaning that differences between adjacent scores (e.g., between 2 and 3) do not represent equal intervals. Accordingly, the reported means are used only as descriptive indicators to summarize the general quality trends observed across case studies, rather than as precise interval-based measurements suitable for statistical inference [21].

Overall, the framework generated between four and twelve files per case study, with an average GQS of 2.40 across all evaluations. This result indicates that most generated artifacts were syntactically valid and functionally coherent, though some required further refinement or configuration adjustments to achieve full compliance. Case studies with explicit code structures and consistent configuration schemas (e.g., ICS, CS2, and CS3) achieved higher GQS averages, while those with sparse or unconventional layouts (e.g., CS5) scored lower due to missing context for patch generation and incomplete retraining logic.

Table 6 presents the detailed practice-level results, combining both case-level and practice perspectives. Each row lists the GQS values for one case study across all applicable practices (PR1–PR15), followed by the per-case average in the final column. The bottom row reports the overall mean GQS per practice, aggregated across all case studies.

Table 5. Generated Files and Average GQS Scores per Case Study

Case Study	Files Generated	Mean GQS
ICS	12	2.29
CS1	8	2.50
CS2	6	2.71
CS3	10	2.52
CS4	4	2.50
CS5	9	1.90
Overall (avg.)	8.2	2.40

The results reveal a consistent pattern of *mid- to high-quality outputs* (typically between 2.5–3.0) for practices related to model versioning (PR3–PR5), evaluation pipelines (PR7), and registry integration, whereas structured metadata logging (PR1–PR2), pinned model loading (PR4), and retraining triggers (PR11) remain partially implemented. CS2 achieved the highest quality (average GQS=2.71), demonstrating mature implementation of model versioning and evaluation components, whereas CS5 exhibited the lowest quality (average GQS=1.90), primarily due to syntax issues and incomplete configuration linkages. Overall, the aggregated mean score of 2.40 confirms that the generated artifacts are generally *functionally valid but still require refinement and parameterization* to reach full compliance with the targeted best practices.

Table 6. Integrated Ground Truth Quality Scores across Case Studies and Practices

Case Study	PR1	PR2	PR3	PR4	PR5	PR6	PR7	PR8	PR9	PR10	PR11	PR12	PR13	PR14	PR15	Avg
ICS	2.0	2.0	3.0	2.0	3.0	3.0	1.0	3.0	3.0	2.0	1.0	–	3.0	2.0	2.0	2.29
CS1	2.0	2.0	3.0	2.0	–	2.5	3.0	3.0	–	2.0	2.5	–	–	2.0	–	2.50
CS2	2.0	3.0	4.0	2.0	3.0	3.0	3.0	2.0	–	2.0	2.0	–	–	–	3.0	2.71
CS3	2.0	2.0	3.5	2.0	2.5	–	3.0	2.5	–	2.0	2.0	–	–	–	3.0	2.52
CS4	2.0	2.0	3.5	2.0	3.0	–	3.0	2.0	–	2.0	–	–	–	–	–	2.50
CS5	2.0	2.0	2.0	1.5	2.0	1.0	2.0	–	–	–	1.5	–	–	–	3.0	1.90
Overall Avg	2.0	2.2	3.3	2.0	2.8	2.9	3.0	2.3	3.0	2.1	2.2	–	3.0	2.0	3.0	2.40

Figure 3 illustrates how the Agentic AI workflow proceeds through detection and generation stages to produce concrete improvements. After the file selection phase, the framework identified *train_hier.py* as relevant to model versioning and registry-related practices. The example highlights how the system transforms incomplete practice implementations into structured, versioned, and testable components. Specifically, in this case, the generation phase replaced simple checkpoint copying logic with an MLflow-based model registration process that logs parameters, metrics, and model versions (PR3/PR5).

7 DISCUSSION

This section discusses how the research questions (RQ1–RQ2) were addressed based on the results presented in Section 6.4. The findings demonstrate the potential and limitations of the Agentic AI architecture using LLMs for analyzing and improving RL pipelines in accordance with best practices.

PR3/PR5 – Model Versioning & Registry*Before modification:*

```
def update_logs(args, log_dir):
    # Copies latest checkpoint; no registry metadata
    dirs = sorted(Path(log_dir).glob('*/*'))
    check = ''
    for item in dirs:
        if "checkpoint" in item.name:
            check = str(item)
    shutil.copytree(check, "checkpoint/")
```

After modification:

```
import mlflow
def register_model(model, version_id, hyperparameters, metrics):
    """Register model in MLflow registry with metadata."""
    with mlflow.start_run() as run:
        mlflow.log_params(hyperparameters)
        mlflow.log_metrics(metrics)
        mlflow.pytorch.log_model(model, f"model_{version_id}")
        model_uri = f"runs/{run.info.run_id}/model_{version_id}"
        mlflow.register_model(model_uri, "MyModelRegistry")
        print(f"Model registered with URI: {model_uri}")
```

Fig. 3. Illustrative concise code excerpts from CS1 (file: `train_hier.py`) showing end-to-end operation of the Agentic AI framework. The generated code replaces checkpoint-only copying with MLflow-based model registration (PR3/PR5).

RQ1: The detection results in Table 4 show that the Agentic AI architecture achieves strong and stable detection of the practices, with an overall F_1 -score of 0.80. Precision (0.82) exceeded recall (0.78), indicating that the system tends to make conservative, evidence-based detections, flagging a practice only when its presence is explicitly confirmed through recognizable structural or semantic indicators. This aligns with the intentional design of the *Detection Agent*, which prioritizes correctness and interpretability over aggressive recall.

The framework was particularly reliable in cases where practices were clearly implemented and isolated within code modules, as in CS3 and CS5 ($F_1 = 1.0$). Conversely, more implicit or distributed implementations, such as those found in the ICS ($F_1 = 0.73$) or in CS2 ($F_1 = 0.67$), challenged the detection. These cases demonstrate that practice visibility, project structure, and code–configuration coupling substantially influence detection accuracy. Nevertheless, even in these more complex environments, the framework successfully captured essential components, such as model saving, evaluation logic, and registry usage, indicating that multi-agent coordination enables robust cross-file and cross-module reasoning.

Overall, the results for RQ1 confirm that an agentic LLM-based architecture can effectively detect the presence or absence of the practices across heterogeneous RL projects.

Beyond numerical performance, these findings also illustrate the architectural benefits of the Agentic AI architecture compared to a single local LLM configuration. The modular, protocol-driven design allows each agent (e.g., File Selection, Detection, Fix Suggestion, Code Generation) to operate with tailored prompts, validation criteria, and reasoning strategies suited to its role. Because agents interact through a structured orchestrator, all exchanges are sequenced, logged, and auditable, ensuring transparency and traceability that a monolithic LLM setup would not provide. Moreover, individual agents can be updated or replaced independently, supporting extensibility toward new practices or domains without redesigning the entire pipeline.

RQ2: To address RQ2, we evaluated the quality of the generated and modified code artifacts using the four-level GQS described in Section 6.3. The results (Table 6) show that the framework produces artifacts of overall good quality, with an aggregated mean GQS of 2.40 across all case studies. This value indicates that most generated artifacts are *functionally valid yet still incomplete*, typically requiring minor refinement or configuration adjustments to achieve full compliance with the targeted best practices. High-quality outcomes were observed for practices related to model versioning (PR3), registry integration (PR5), and evaluation pipelines (PR7), where scores consistently reached or exceeded 3.0. These areas benefit from more explicit structural patterns (e.g., MLflow calls, evaluation callbacks, or registry APIs) that the framework can reliably reproduce.

In contrast, practices involving structured metadata logging (PR1–PR2), pinned model loading (PR4), and retraining triggers (PR11) achieved lower scores (around 2.0–2.2), reflecting partial implementation, missing contextual information, or limited configuration linkage. These findings indicate that while the framework can autonomously generate coherent and semantically meaningful code, its effectiveness still depends on the clarity and completeness of the initial detection evidence as well as the inherent complexity of the targeted practice.

While the Agentic AI architecture introduces some overhead in coordination latency and implementation complexity, its modularity and specialization offer clear advantages for sustainable evolution. New agents can be seamlessly integrated for additional best practices or emerging RL tasks, and validation gates can be extended without disrupting the workflow. Thus, although it does not remove inherent LLM limitations, the architecture provides a scalable, extensible, and auditable foundation for analysis that a local, single-model approach could not easily achieve.

8 THREATS TO VALIDITY

In this section, we discuss the potential threats to validity and the steps taken to mitigate them in the design, execution, and interpretation of this study.

Internal validity. To ensure internal validity, all practices were consistently applied across case studies using a structured evaluation protocol. The detection, suggestion, and generation phases were validated independently, and all generated files were syntax-checked and manually reviewed to mitigate errors or hallucinations. Agent configurations (File Selection, Detection, Suggestion, Code Generation, Validation) remained fixed throughout, ensuring that observed differences stemmed from project characteristics rather than uncontrolled system variation.

External validity. We mitigated this threat by applying the framework to six diverse case studies, including an industrial RL pipeline and multiple open-source RL projects with varying architectures, dependencies, and configurations. This heterogeneity increases the representativeness of our results across real-world scenarios. Nevertheless, we acknowledge that the case studies mainly focus on Python-based RL systems using frameworks such as Stable-Baselines3 and MLflow. Extending the validation to other ecosystems (e.g., Ray RLLib, CleanRL, or proprietary toolchains) and additional lifecycle phases (e.g., deployment monitoring or online adaptation) would further improve generalizability.

Construct validity. In our case, construct validity is tied to the operationalization of practice compliance and generation quality. We mitigated threats by grounding the practice definitions in an extensive literature review and standards on RL engineering. Each GQS level was supported by explicit operational criteria, and multiple evaluators cross-checked the assigned scores to minimize subjectivity. Different executions or model updates may lead to variations in detection or generation behavior. Although using GPT-4o ensured consistent high-quality reasoning and context retention, prompt sensitivity remains a known limitation. We mitigated this by designing structured, task-specific prompts and fixed

conversation templates for all agents. However, differences in prompt interpretation or output randomness may still introduce variability, which should be addressed by integrating deterministic reasoning modes and caching strategies in future iterations.

9 CONCLUSIONS AND FUTURE WORK

This work introduced an Agentic AI architecture that coordinates multiple LLM-based agents to detect, reason about, and automatically improve compliance with best practices in RL pipelines. The architecture operates in four coordinated stages: file selection, detection, suggestion, and code generation, supported by structured prompts and shared context management. Our evaluation across six case studies demonstrated that the Agentic AI architecture can reliably identify best-practice implementations ($F_1 = 0.80$), but while the generated artifacts were largely syntactically and semantically valid (overall score = 2.40), they were often *functionally correct yet still incomplete*, typically requiring minor refinements or configuration adjustments to achieve full compliance with the targeted best practices.

The key contribution of this work is to demonstrate that an Agentic AI, LLM-based architecture can not only detect compliance issues but also synthesize meaningful and valid code modifications. The combination of diverse reasoning, structured communication, and validation checkpoints enables traceable and explainable improvements that go beyond static analysis or heuristic checking.

Future work will focus on several directions. We plan to expand the practice catalog and evaluation to additional domains, including secure RL, multi-agent coordination, and explainable decision-making pipelines. Additionally, to integrate continuous validation agents into MLOps pipelines, enabling real-time detection and correction of practice deviations.

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] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*. 265–283.
- [2] Lorenzo Canese, Gian Carlo Cardarilli, Luca Di Nunzio, Rocco Fazzolari, Daniele Giardino, Marco Re, and Sergio Spanò. 2021. Multi-agent reinforcement learning: A review of challenges and applications. *Applied Sciences* 11, 11 (2021), 4948.
- [3] Hugh Chen, Scott Lundberg, and Su-In Lee. 2017. Checkpoint ensembles: Ensemble methods from a single training process. *arXiv preprint arXiv:1710.03282* (2017).
- [4] Santhosh Chitraju Gopal Varma. 2024. Optimizing Machine Learning Pipelines: Best Practices for Scalable and Efficient Model Deployment. *Available at SSRN 5226791* (2024).
- [5] Behrouz Derakhshan, Alireza Rezaei Mahdiraji, Tilmann Rabl, and Volker Markl. 2019. Continuous Deployment of Machine Learning Pipelines.. In *EDBT*. 397–408.
- [6] Gabriele Digregorio, Marco Di Gennaro, Stefano Zanero, Stefano Longari, and Michele Carminati. 2025. When Secure Isn't: Assessing the Security of Machine Learning Model Sharing. *arXiv preprint arXiv:2509.06703* (2025).
- [7] Raileanu R. Eimer T., Lindauer M. 2023. Hyperparameters in Reinforcement Learning and How to Tune Them. In *International Conference on Machine Learning*. 9104–9149.
- [8] Assaf Eisenman, Kiran Kumar Matam, Steven Ingram, Dheevatsa Mudigere, Raghuraman Krishnamoorthi, Murali Annavam, Krishnakumar Nair, and Misha Smelyanskiy. 2020. Check-n-run: A checkpointing system for training recommendation models. *arXiv preprint arXiv:2010.08679* 5 (2020).
- [9] B Eken, S Pallewatta, NK Tran, A Tosun, and MA Babar. [n. d.]. A Multivocal Review of MLOps Practices, Challenges and Open Issues. *arXiv 2024. arXiv preprint arXiv:2406.09737* ([n. d.]).

- [10] Toluwase Peter Gbenle, Abraham Ayodeji Abayomi, Abel Chukwuemeke Uzoka, Oyejide Timothy Odofoin, Oluwasanmi Segun Adanigbo, and Jeffrey Chidera Ogeawuchi. 2024. Developing an AI Model Registry and Lifecycle Management System for Cross-Functional Tech Teams. *International Journal of Scientific Research in Science, Engineering and Technology* 11, 4 (2024), 442–456. <https://doi.org/10.32628/IJSRSET25121179>
- [11] Pablo Hernandez-Leal, Bilal Kartal, and Matthew E Taylor. 2019. A survey and critique of multiagent deep reinforcement learning. *Autonomous Agents and Multi-Agent Systems* 33, 6 (2019), 750–797.
- [12] Shengyi Huang, Rousslan Fernand Julien Dossa, Chang Ye, Jeff Braga, Dipam Chakraborty, Kinal Mehta, and JoÃGo GM AraÃsjo. 2022. Cleanrl: High-quality single-file implementations of deep reinforcement learning algorithms. *Journal of Machine Learning Research* 23, 274 (2022), 1–18.
- [13] Naveen Krishnan. 2025. Advancing multi-agent systems through model context protocol: Architecture, implementation, and applications. *arXiv preprint arXiv:2504.21030* (2025).
- [14] Valliappa Lakshmanan, Sara Robinson, and Michael Munn. 2020. *Machine Learning Design Patterns*. O’Reilly Media, Inc.
- [15] Van-Duc Le, Cuong-Tien Bui, and Wen-Syan Li. 2023. Veml: An end-to-end machine learning lifecycle for large-scale and high-dimensional data. *arXiv preprint arXiv:2304.13037* (2023).
- [16] Donghwan Lee, Niao He, Parameswaran Kamalaruban, and Volkan Cevher. 2020. Optimization for reinforcement learning: From a single agent to cooperative agents. *IEEE Signal Processing Magazine* 37, 3 (2020), 123–135.
- [17] Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. 2018. Hyperband: A novel bandit-based approach to hyperparameter optimization. *Journal of Machine Learning Research* 18, 185 (2018), 1–52.
- [18] Peizheng Li, Jonathan Thomas, Xiaoyang Wang, Ahmed Khalil, Abdelrahim Ahmad, Rui Inacio, Shipra Kapoor, Arjun Parekh, Angela Doufexi, Arman Shojaeifard, and Robert J. Piechocki. 2022. RLOps: Development Life-Cycle of Reinforcement Learning Aided Open RAN. *IEEE Access* 10 (2022), 113808–113826. <https://doi.org/10.1109/access.2022.3217511>
- [19] Eric Liang, Richard Liau, Robert Nishihara, Philipp Moritz, Roy Fox, Ken Goldberg, Joseph Gonzalez, Michael Jordan, and Ion Stoica. 2018. RLlib: Abstractions for distributed reinforcement learning. In *International conference on machine learning*. PMLR, 3053–3062.
- [20] Hui Miao, Ang Li, Larry S Davis, and Amol Deshpande. 2016. Modelhub: Towards unified data and lifecycle management for deep learning. *arXiv preprint arXiv:1611.06224* (2016).
- [21] Geoff Norman. 2010. Likert scales, levels of measurement and the “laws” of statistics. *Advances in health sciences education* 15, 5 (2010), 625–632.
- [22] Evangelos Ntontos, Stephen J. Warnett, and Uwe Zdun. 2025. Rule-Based Assessment of Reinforcement Learning Practices Using Large Language Models. In *4th International Conference on AI Engineering ? Software Engineering for AI (CAIN)*. <https://doi.org/10.1109/CAIN66642.2025.00009>
- [23] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32 (NeurIPS 2019)*. 8024–8035.
- [24] Antonin Raffin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus, and Noah Dormann. 2021. Stable-baselines3: Reliable reinforcement learning implementations. *Journal of machine learning research* 22, 268 (2021), 1–8.
- [25] Mohammad Reza Samsami and Hossein Alimadad. 2020. Distributed deep reinforcement learning: An overview. *arXiv preprint arXiv:2011.11012* (2020).
- [26] Marius Schlegel and Kai-Uwe Sattler. 2022. Management of Machine Learning Lifecycle Artifacts: A Survey. *arXiv:2210.11831 [cs.DB]* <https://arxiv.org/abs/2210.11831>
- [27] Ruchi Sharma and Kiran Davuluri. 2019. Design patterns for Machine Learning Applications. In *2019 3rd International Conference on Computing Methodologies and Communication (ICCMC)*. 818–821.
- [28] Klaas-Jan Stol, Paul Ralph, and Brian Fitzgerald. 2016. Grounded theory in software engineering research: a critical review and guidelines. In *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*. IEEE, 120–131.
- [29] Shashi Thota, Subrahmanyasarma Chitta, V Alluri, V Vangoor, and Chetan Sasidhar Ravi. 2022. MLOps: Streamlining machine learning model deployment in production. *African J. of Artificial Int. and Sust. Dev* 2, 2 (2022), 186–206.
- [30] Hironori Washizaki, Foutse Khomh, Yann-Gaël Guéhéneuc, Hironori Takeuchi, Naotake Natori, Takuo Doi, and Satoshi Okuda. 2022. Software-Engineering Design Patterns for Machine Learning Applications. *Computer* 55, 3 (2022), 30–39.
- [31] Doris Xin, Hui Miao, Aditya Parameswaran, and Neoklis Polyzotis. 2021. Production machine learning pipelines: Empirical analysis and optimization opportunities. In *Proceedings of the 2021 international conference on management of data*. 2639–2652.
- [32] Kaiqing Zhang, Zhuoran Yang, and Tamer Başar. 2021. Multi-agent reinforcement learning: A selective overview of theories and algorithms. *Handbook of reinforcement learning and control* (2021), 321–384.
- [33] Yue Zhou, Yue Yu, and Bo Ding. 2020. Towards mlops: A case study of ml pipeline platform. In *2020 International conference on artificial intelligence and computer engineering (ICAICE)*. IEEE, 494–500.