Rule-Based Assessment of Reinforcement Learning Practices Using Large Language Models

Evangelos Ntentos*, Stephen John Warnett*†, Uwe Zdun*
*Faculty of Computer Science, Research Group Software Architecture, University of Vienna, Vienna, Austria
†UniVie Doctoral School Computer Science DoCS, University of Vienna, Vienna, Austria
firstname.lastname@univie.ac.at

***Abstract*—In the fast-evolving field of artificial intelligence, Reinforcement Learning (RL) plays a crucial role in developing agents that can make decisions. As these systems become increasingly complex, the need for standardized and automated training methods becomes apparent. This paper presents a rule-based framework that integrates Large Language Models (LLMs) and heuristic-based code detectors to ensure compliance with best practices in RL training pipelines. We define a set of architectural rules that target best practices in important areas of RL-based architectures, such as checkpoints, hyperparameter tuning, and agent configuration. We validated our approach through a large-scale industrial case study and ten open-source projects. The results show that LLM-based detectors generally outperform heuristic-based detectors, especially when handling more complex code patterns. This approach effectively identifies best practices with high precision and recall, demonstrating its practical applicability.**

***Index Terms*—Reinforcement Learning, Best Practices, Machine Learning, Architecture Rules, Case Studies, LLMs**

I. Introduction

In RL systems, an agent's success in reaching its goals relies heavily on the training strategies used [17]. These strategies include balancing exploration and exploitation, shaping rewards, and applying algorithms such as Q-learning or policy gradients for policy optimization [13], [10]. Effective training methods are essential to improving learning efficiency and overall system performance.

Several common best practices or patterns are part of these training methods. For example, checkpoints [7], [2] are used to save the model's state at regular intervals during training, allowing recovery from interruptions, which helps maintain efficiency. Similarly, hyperparameter tuning [3] involves adjusting parameters like learning rates and batch sizes. Tuning their values can increase performance. Single-agent versus multi-agent training [14], [5], [19] could be considered based on the complexity of a task. A single agent is a simpler approach, but it has limitations when scaling up. At the same time, with multi-agents, behavior can be competitive or collaborative and is more adaptive for environments that are dynamic or subject to possible changes.

We developed a rule-based approach that uses LLMs to address the challenge of assessing such best practices in RL training pipelines. Our approach defines 31 architectural rules to evaluate RL training practices, realizes an approach and flow-based software architecture for LLM-based detection, and provides LLM-based detection components for each of the 31 rules. Additionally, we compare them with typical heuristic-based code detectors, which are mainly focused on straightforward code patterns.

Heuristic-based detectors are effective for quickly identifying common code structures, such as specific operations for checkpoint implementations or function calls. However, they struggle with identifying more intricate or non-standard code structures or code in polyglot settings where multiple languages or technologies are used. This challenge is where LLM-based detection can become valuable, as it can interpret the broader context and intent behind the code or deal with unforeseen code structures.

In complex case studies, combining both approaches can be particularly effective. For instance, heuristic-based code detectors can detect common code patterns, while LLMs can provide deeper analysis where needed, particularly when rules are implemented in less direct ways. Using these detection techniques, our approach can ensure that RL training strategies are aligned with best practices, even with complex or non-standard code structures or in polyglot environments.

This paper aims to answer the following research questions:

- **RQ1**: How effectively can LLMs assess best practices in RL training pipelines compared to heuristic-based code detectors?
- **RQ2**: What are the key architectural rules required for compliance with the best practices for checkpoints, hyperparameter tuning, training/agent configuration, and single vs. multi-agent RL in RL-based systems?

We tested the effectiveness of our approach through a large-scale industrial case study that uses RL in a Cyber-Physical System (CPS) for production automation, as well as ten open-source RL systems. The industrial case study aims to monitor several CI/CD pipelines across production facilities. This comprehensive evaluation allowed us to assess the successful implementation of our approach in different RL environments and training workflows.

The paper is organized as follows: Section II introduces RL practices. Related work is described in Section III. The methodology and specifics of automatic rule checking are discussed in Section IV. Case studies are presented in Section V. The quantitative evaluation and case study are discussed in Section VI. Section VII interprets the results, while Section VIII addresses potential validity threats. Section IX concludes the paper.

II. Background on Reinforcement Learning Practices

This section provides background on the RL practices that are central to this work, specifically covering *checkpoints*, *hyperparameter tuning*, and *training and agent configuration*.

Checkpoints [7], [2] are essential in RL, as they save the agent's learning state at intervals. Checkpoints allow training to resume from specific points, protecting against data loss and allowing adjustments without needing to restart from scratch. Regular checkpoints also enable progress tracking and fine-tuning based on intermediate results, which is key to improving the learning process.

Hyperparameter tuning [3] is an important practice in RL. The success of RL algorithms often relies on choosing the right hyperparameters, such as learning rates and discount factors. Adjusting these settings correctly can significantly increase the speed and effectiveness of the learning process. Popular methods for tuning include grid search, random search, and more sophisticated approaches like Bayesian optimization.

A key difference in RL is between *single-agent* [5] and *multi-agent* [14], [20] systems. Single-agent setups involve one agent learning to make the best decisions by interacting with its environment. In contrast, multi-agent systems consist of several agents that might work together or against each other, adding layers of complexity to the learning process. Multi-agent RL (MARL) faces extra challenges like non-stationarity, where the environment evolves as other agents learn and take action, demanding more sophisticated strategies.

III. 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. [8] investigate the evolution of RL algorithms, pointing out the transition from single-agent systems to multi-agent configurations that depend on distributed optimization. Canese et al. [1] analyze multi-agent algorithms, emphasizing crucial elements like scalability, non-stationarity, and observability, which are all vital in multi-agent RL contexts.

Eimer et al. [3] provide an in-depth analysis of hyperparameter tuning approaches, covering techniques like grid search, random search, and Bayesian optimization. Li et al. [9] propose the Hyperband method for hyperparameter optimization, which efficiently reduces computational demands while identifying optimal parameters. Our work builds on top of these studies by incorporating automated detection of hyperparameter tuning in RL using a rule-based assessment that leverages LLMs and heuristics.

Chen et al. [2] explore the use of checkpoints in training processes, presenting methods to improve model performance by periodically saving and validating models. Similarly, Eisenman et al. [4] explore checkpointing systems in training recommendation models, outlining how checkpoints can be utilized to boost training efficiency. Our approach aligns with these studies by emphasizing the significance of checkpoints. Still, it additionally applies automation of the checkpoint validation process directly through source code, using LLM-based and heuristic-based analysis of specific architectural rules.

Hernandez-Leal et al. [6] 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 formulating rules relevant to multi-agent training and validating them using LLM-based and heuristic-based methods.

Zhang et al. [19] 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 rules for multi-agent systems and validating their use in real-world systems.

Schneider et al. [12] present a rule-based approach focusing on verifying compliance related to security in microservice architectures. While their work centers on security, the rule-based detection system they describe shares similarities with our approach. However, our approach combines rule-based detection with LLM-based and heuristic-based methods to ensure RL systems adhere to best practices.

IV. Approach

This section describes the research methods followed in this study and our rule-based assessment approach, which uses LLMs and heuristics to ensure conformance to best practices in RL training code. The data used in and produced as part of this study have been made available online for reproducibility[1].

A. Research Methods

Figure 1 outlines the research methodology employed in this study. We initially reviewed various knowledge sources on RL-specific best practices, including research papers[15], [11], [18], [3], [7], web resources and open-source repositories. Subsequently, we conducted a qualitative analysis using Grounded Theory coding methods [16], such as open and axial coding, to analyze the collected data and extract relevant practices (described in Section II). We closely followed established pattern catalogs, such as those by Lakshmanan [7], for comprehensive insights into ML concerns. We then, based on this analysis, formulated 31 architectural rules and developed the corresponding heuristic-based detectors and LLM-based detectors. Moreover, we analyzed the case studies using the detectors and we evaluated our approach measuring the precision, recall and F1 score metrics. More specifically, the main steps after data analysis were:

Step 1. Definitions of Architectural Rules: In the first step, we identified key aspects of RL training that focus on checkpoints, hyperparameter tuning, and training and agent configuration. Our approach established specific rules to ensure consistent and correct application of best practices in RL training code. For example, rules were set for saving model states at regular intervals with metadata for integrity

[1]https://doi.org/10.5281/zenodo.14051965

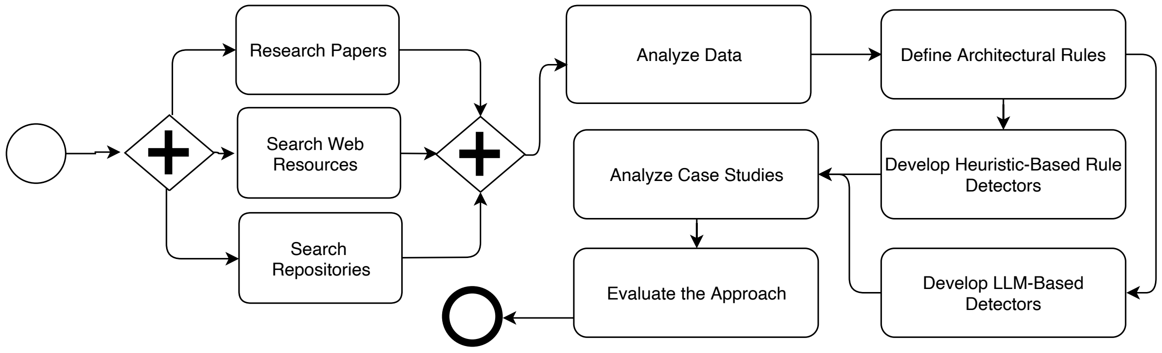

Fig. 1. Overview of the research method followed in this study

checks. Hyperparameter tuning rules were defined to validate and adjust parameters essential for performance. Furthermore, additional rules focus on training configurations, managing multi-agents to ensure compatibility, and clearly defining state and action spaces.

Step 2. Development of Heuristic-Based Code Detectors: In this step, we designed a set of heuristic-based code detectors that strictly comply with the rules described in Step 1. These detectors parse the RL training code and verify its compliance with the architectural rules. Every code detector is created to recognize practices such as checkpoints, hyperparameter tuning, and agent configurations. The detectors function within a service that combines the outputs of these automatic detections to create structured reports in JSON format.

Step 3. Development of LLM-Based Detector: We developed a framework that uses LLMs to handle specifications within the RL training pipeline. This framework enables users to define queries without writing extensive code, as the LLMs interpret these descriptions and generate corresponding answers. By utilizing LLMs, the framework can adapt to project-specific requirements and complex representations of RL training practices. The LLM-based detection system can work in addition to the heuristic-based code detectors. This approach simplifies the process for non-expert users while maintaining rigorous validation standards.

Step 4. Application to Case Studies and Validation: The final step of our approach involved applying the heuristic-based code detectors and the LLM-based detectors to a number of RL systems for evaluation. This process included extensive validation across one industrial case study and ten open-source RL systems.

B. Architecture of the Heuristic-Based Code Detection Approach

Heuristic code-based detector consists of several key modules, each contributing to validating RL training practices. A central service serves as an *Orchestrator*, processing the source code and managing the analysis workflow to ensure that all necessary validation tasks are executed. *Rule-Based Detectors* are used to assess the RL code against best practices, with *Heuristic-Based Detectors* identifying general patterns and common issues. The *Code Analyzer* parses the source code, applies these detectors, and evaluates the code's compliance with best practices, identifying any rule violations. Once the analysis is complete, the *Result Report* component compiles the findings into JSON reports. This service functions within an *MLOps Pipeline* or as a standalone tool. Figure 2 visually represents the system architecture.

C. Architecture of LLM-Based Detection Approach

"Does the Python file reinforcement learning project contain checkpoints? Look for specific keywords or phrases that indicate a checkpoint is being created. For instance, if a "save" on a model object is found, that could be a checkpoint. Or if code contains the phrase "checkpoint".

The LLM-based detection approach is designed to automate the detection of best practices in RL-based systems. The architecture shown in Figure 3 works in the same context as the heuristic-based approach, i.e., it can also be used standalone or as part of an *MLOps Pipeline*. It also generates a detailed *Result Report*. The core component, *LLM Code Flow Executor*, manages the execution of the assessment process or a specific validation task. It runs the workflow, defined via the *LLM Code Flows* component, to execute the required detection and validation steps. *LLM* is a component that abstracts and wraps different supported LLMs. It defines LLM parameters such as model name and token limits, enabling the use of various models, including GPT-3.5, GPT-4o, or Llama. *LLM*'s class *Conversation* handles the interactions with the LLMs, keeping track of ongoing conversations and guiding the code detectors based on the context. These conversations are used in the *LLM Code Flows*.

The *LLM-based Detectors* component realizes all detection tasks, and it contains classes to realize the different rules described in this paper, such as those of checkpoints and hyperparameter tuning. The code flow provides a Data Transfer Object, *LLM Code Flow DTO*, to each executed detector in the workflow. It carries essential information such as project metadata, source directories, and validation status reports. As this object is passed through various filters (i.e., different detectors), it creates a report of the detected rules incrementally.

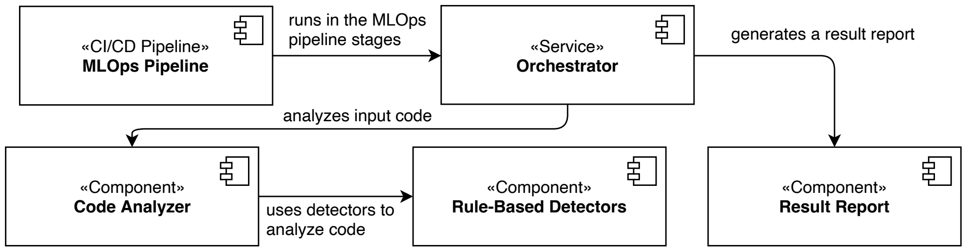

Fig. 2. Architecture of the Heuristic-Based Detector

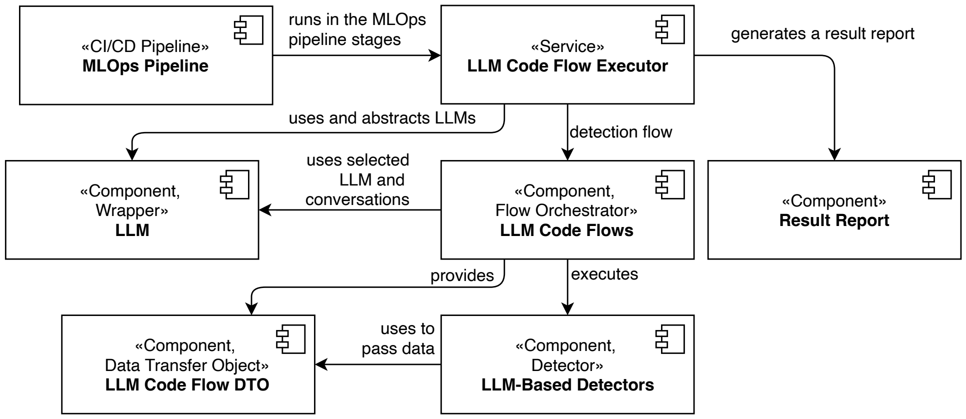

Fig. 3. Architecture of the LLM-Based Detector

D. Rules and Detectors

*a) **Rules**:* Table I outlines the rules defined in this study. Each rule is binary, meaning it can either be true or false. A rule is true if the corresponding architectural guideline is observed in the application's source code. Conversely, a rule is false if the source code contradicts the architectural rule or lacks relevant insights.

In the example shown in Figure 4, the RL code for creating a checkpoint adheres to the rule "R02: Implement adjustable checkpoint intervals to accommodate varying training durations and system performance", as this parameter is directly implemented within the *CheckpointCallback* method. In contrast, the architectural rule "R11: Define all hyperparameters clearly and categorize them based on functionality and model components." does not hold because there are no indications of defining and grouping hyperparameters.

*b) **Heuristic-Based Code Detectors**:* The detectors were implemented using Python, with regular expressions and pattern-matching techniques to identify specific code structures and consist of functions designed to analyze code and verify compliance with architectural rules. Using regular expressions and parsing methods, they identify patterns like agent configuration or environment management. For example, in RL code that creates checkpoints (see Figure 4), heuristic-based detectors look for keywords like *checkpoint* or *.save()* to identify general patterns or more concrete keywords to identify saving frequency (*save_frequency*) and saving directory (*checkpoint_directory*). However, they may mistakenly capture unrelated functions.

*c) **LLM-Based Detectors**:* LLM-based detectors use LLMs to inspect source code, spot patterns, and check if specific rules are followed. Unlike traditional rule-based methods, these detectors interpret code more flexibly, making it possible to find patterns that don't strictly fit set rules. Our LLM-Based Detectors are highly modular, meaning each checks a specific occurrence of code fragments, realizing a part of a rule rather than checking multiple aspects simultaneously. This modularity helps to, firstly, get precise and relatively predictable results from the LLMs and, secondly, establish traceability between the rule and the places in the source code where the code fragments realizing the rule occur.

For example, when dealing with checkpoints, the LLM gets instructed to examine the code for 'save()' functions or keywords like 'checkpoint' and then determines if they align with the specified rules. In the RL code shown in Figure 4, the LLM would recognize key parts of the code, such as 'model.save()' calls, and assess if they meet rule R02 by considering how checkpoint intervals are managed.

LLM-based detectors provide more detailed analysis in cases where more detailed detection is needed, such as checking that checkpoints are saved in the right folder. They can track the creation of directories and connect it to checkpoint storage, as shown in the custom detection example in Figure 4. This analysis ensures that even complex rules are followed correctly.

The detection process follows several steps: first, the LLM-based approach checks the code for possible issues; then, it compares what it finds with the set architectural rules; and finally, it organizes the results in a clear format. This method provides a complete review of the code, giving insights into rule compliance and improving validation accuracy. Please note that the LLM is only one of many components required to realize these steps.

Figure 5 shows a conversation template that establishes a process for interacting with an LLM to review the Python code of an RL project, specifically focusing on the presence of checkpoint functionalities[2]. It begins by setting up a system message that instructs the assistant to assist a junior developer in analyzing RL code. The assistant is tasked with searching for checkpoint-related code in the provided Python file, with instructions on how to respond if such code is identified. The script includes an example of Python code demonstrating checkpoint functionality, along with a sample response that highlights the relevant sections. The corresponding function constructs the conversation by combining the system message, a checkpoint-related query, the example code and response, and the target code to be analyzed. This structured approach, with clear instructions and examples, helps improve the accuracy and relevance of the LLM's analysis of checkpoint implementations in the target Python code.

```
from stable_baselines3.common.callbacks import CheckpointCallback

checkpoint_directory = "checkpoints"
os.makedirs(checkpoint_dir, exist_ok=True)

checkpoint_callback = CheckpointCallback(
    save_frequency=1000,  # Save the model every 1000 steps
    save_path=checkpoint_directory,
    name_prefix=env.unwrapped.metadata.get('name'))
model.save(f"{env.unwrapped.metadata.get('name')}_
                              {time.strftime('%Y%m%d-%H%M%S')}")
```

Fig. 4. Source Code Example of Checkpoint Implementation (Taken from CS1)

V. Case Studies

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

A. Industrial Case Study

To validate the effectiveness of our rule-based assessment approach, we applied our *ML pipeline insights service* to a comprehensive industrial case study. This case study revolves around a sophisticated single-agent and MARL framework that facilitates extensive testing and optimization of AI policies across diverse environments. The industrial system, developed by a global provider of production automation solutions, is an advanced platform that features several key components:

- **Agent Management Modules**: These modules enable efficient simulation of agent behaviors and learning processes, allowing for the detailed study of multi-agent interactions and dynamics.
- **Customizable Training Environments**: Tools within the system allow for precise adjustments to training scenarios, creating varied learning environments that can be tailored to specific strategic tasks.
- **Centralized and Decentralized Learning Approaches**: The platform supports centralized and decentralized learning, optimizing training effectiveness through adaptable policy definitions and performance tuning.
- **Hyperparameter Tuning and Optimization Strategies**: The system includes advanced capabilities for hyperparameter tuning and employs sophisticated optimization strategies crucial for refining AI behaviors and achieving peak performance.
- **Production Automation Application**: The MARL framework is used for automating production processes, with components such as sophisticated simulators, AI optimizers, and training and inference modules.

The *Heuristic code-based detector* is integrated into the MLOps pipelines of this system, using a service-based architecture to provide real-time feedback on the training and inference processes. This integration ensures that the training methods follow best practices, improving the reliability and performance of AI agents.

The *LLM-based detector* is also embedded within the MLOps pipelines of this system using a framework to assess RL training practices implemented in the system. This setup enables more nuanced detection of best practices by interpreting complex code patterns and natural language specifications.

The platform was created by a diverse team of experts and is intended for use on factory shop floors. It aims to standardize and improve production processes by utilizing the advanced MARL framework to address different operational challenges, ultimately boosting efficiency. For this reason, it is crucial for ML specialists and software architects to have a standardized approach to gaining insights into software architectures across different customer projects, especially within the context of their MLOps pipelines. Through this industrial case study, we aim to demonstrate the real-world application and effectiveness of our rule-based assessment tool.

B. Open-source Case Studies

The open-source case studies in Table II (CS1 - CS10) cover different 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

[2]See detailed conversations and prompts in LLM-Based Detectors source files in https://doi.org/10.5281/zenodo.14051965

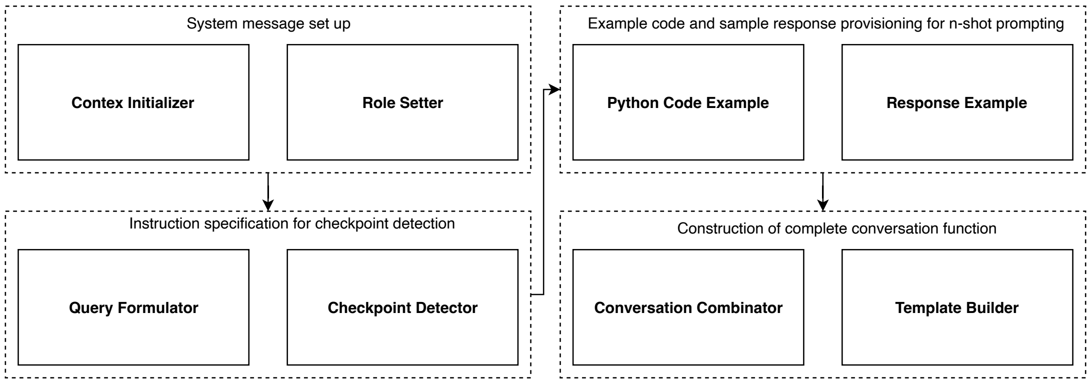
System message set up
Contex Initializer
Role Setter
Example code and sample response provisioning for n-shot prompting
Python Code Example
Response Example
Instruction specification for checkpoint detection
Query Formulator
Checkpoint Detector
Construction of complete conversation function
Conversation Combinator
Template Builder

Fig. 5. Conversation Template of the LLM-Based Detector for Checkpoints

TABLE I
ARCHITECTURAL RULES FOR REINFORCEMENT LEARNING SYSTEM ARCHITECTURES

Rule	**Description**
Rules on Checkpoints	
R01	Maintain a single, dedicated directory for storing all checkpoints.
R02	Implement adjustable checkpoint intervals to accommodate varying training durations and system performance.
R03	Each checkpoint must include detailed metadata such as environment specifics, training steps, and timestamp.
R04	Validate the integrity and completeness of each checkpoint post-save using checksums or similar methods.
R05	Ensure that checkpoints are fully compatible with the training policies used, including versions and configurations.
R06	Maintain a history of checkpoints based on configurable retention policies.
R07	Implement mechanisms to compress or selectively store only significant differences between consecutive checkpoints.
R08	Regularly test the recoverability of checkpoints in a controlled environment.
R09	Automate the cleanup of outdated checkpoints based on retention policies.
R10	Checkpoints are created without considering the dependencies or the state of the training environment.
Rules on Hyperparameter Tuning	
R11	Define all hyperparameters clearly and categorize them based on functionality and model components.
R12	Validate hyperparameters before usage to ensure they fall within reasonable and effective ranges.
R13	Employ evolutionary algorithms to explore the parameter space systematically.
R14	Tune hyperparameters considering their interdependencies.
R15	Implement automated tuning systems like grid search, random search, or Bayesian optimization.
R16	Integrate continuous hyperparameter tuning within the training process.
R17	Monitor the impact of hyperparameter changes on model performance continuously.
R18	Apply version control practices to hyperparameter tuning experiments.
R19	Optimize resource allocation during hyperparameter tuning.
R20	Conduct sensitivity analyses to identify impactful hyperparameters.
Rules on Training and Agent Configuration	
R21	Maintain a clear distinction between training parameters and hyperparameters.
R22	Include configurable training options like total timesteps, n_steps, and epochs.
R23	Use consistent seeding across all random number generators.
R24	Track key performance metrics like rewards, losses, and evaluation scores.
R25	Configure individual policies for each agent based on specific roles or tasks.
R26	Ensure compatibility across all agents and their policies.
R27	Define consistent observation/action spaces for all agents.
R28	Track performance metrics for each agent individually.
R29	Assign unique identifiers to each agent.
R30	Define a shared interactive environment for all agents.
R31	Clearly define the state and action spaces for each agent.

environments. Some systems are designed to be flexible, enabling dynamic interactions in varying settings. Many of these systems also aim to enhance decision-making by employing techniques like action masks in PPO, which assist in managing restricted or undesirable actions in specific situations.

VI. VALIDATION

In this section, we validate our approach by demonstrating the applicability and effectiveness of the defined rules in ensuring best practices in RL systems. Our validation uses the larger industrial case study and ten open-source case studies of RL systems as subjects, introduced in Section V. This comprehensive evaluation allows us to assess the practical

TABLE II
OVERVIEW OF PYTHON-BASED SYSTEMS AND THEIR FUNCTIONALITIES

ID	*Description*
ICS	This system focuses on an advanced framework for single-agent and MARL. It enables thorough testing and optimization of AI policies in diverse environments (See Section V-A for details).
CS1	In this system, zombies originate at the top border of the screen and move downward along varying, unpredictable trajectories until they reach the bottom border. https://pettingzoo.farama.org/tutorials/sb3/kaz/
CS2	In this system, Waterworld, the simulation revolves around archaea organisms navigating their environment in a quest for survival. https://pettingzoo.farama.org/tutorials/sb3/waterworld/
CS3	This system is a Chess example. It uses the observation and action spaces similar to the AlphaZero method, with slight modifications. https://pettingzoo.farama.org/tutorials/sb3/connect_four/
CS4	This system is a game that uses a physics-based challenge where the objective is to guide a ball to the left wall of the game's boundary. https://pettingzoo.farama.org/tutorials/rllib/pistonball/
CS5	This system is a game where two players must connect four of their tokens vertically, horizontally, or diagonally. https://docs.agilerl.com/en/latest/tutorials/pettingzoo/dqn.html
CS6	This system is a classic Atari game, where there are two ships controlled by two players who are each trying to maximize their score. https://pettingzoo.farama.org/tutorials/agilerl/MADDPG/
CS7	This system trains AI agents to play Tic-Tac-Toe using the PettingZoo environment. https://github.com/Farama-Foundation/PettingZoo/blob/master/tutorials/Tianshou/3_cli_and_logging.py
CS8	In this system, there are two agents: the 'speaker' and the 'listener'. The 'speaker' agent possesses the ability to communicate verbally but lacks the capability to move autonomously.https://docs.agilerl.com/en/latest/tutorials/pettingzoo/matd3.html#matd3-tutorial
CS9	This system is also a classic Atari game similar to CS6. https://github.com/vwxyzjn/cleanrl/blob/master/cleanrl/ppo_pettingzoo_ma_atari.py
CS10	This system is similar to CS1. It trains agents in the "Knights-Archers-Zombies" environment using the Black Death wrapper to handle agent deaths effectively. https://pettingzoo.farama.org/tutorials/sb3/kaz/

implementation of our rules across diverse RL environments and training practices.

A. Validation Setup

The ground truth for the validation was established by manually inspecting the source code of each case study and annotating the presence or absence of the architectural rules. This manual inspection was conducted by the authors and cross-verified by RL experts to ensure accuracy

The validation was conducted by applying our two kinds of detectors across multiple case studies. The source code of each RL system was analyzed, and the detectors were used to verify whether the code adhered to the defined rules (see Table I). This setup allowed for a comprehensive and accurate validation of our approach using heuristic-based and LLM-based detection methods.

GPT-3.5 Turbo, version 0613, was selected to validate our LLM-based detectors due to its robust performance, making it suitable for analyzing complex code patterns in RL training pipelines. It balances computational efficiency and performance, offering high accuracy and flexibility without the significantly higher costs associated with larger models such as GPT-4 or GPT-4o. In addition, GPT-3.5 Turbo is currently in LLM benchmarks close to the best non-commercial LLMs[3]. These factors are important to consider as many organizations do not want to or cannot use cloud-hosted, commercial LLMs. Thus, we opted for GPT-3.5 Turbo as there are equally good open alternatives.

We employed standard evaluation metrics to measure the accuracy of our approach. For measuring the accuracy, True Positives (TP) are instances where the rule-checking mechanism correctly identifies a source code instance as compliant with the best RL training practices. False Positives (FP) are instances where a source code instance is identified as compliant when it is not. False Negatives (FN) are instances where the method fails to find a compliant code instance. True Negatives (TN) are not considered in our validation, as our ground truth inspection only marks up the TPs, not all true negative code instances. We use the following metrics to measure accuracy:

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

$$\text{F1-score} = 2 \times \left(\frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \right)$$

B. Results

In the remainder of this section, we present the validation results, summarized in Tables III and IV, assessing the accuracy of the rule-based detectors.

1) Industrial Case Study Results: In Table III, we observe the performance of both detectors across the architecture rules. For rules on checkpoints, the *LLM-based detectors* identified seven true positives and missed only one, achieving a precision of 1.0, recall of 0.88, and an F1 score of 0.933, indicating high effectiveness in accurately detecting checkpoints. In contrast, the *heuristic-based detectors* detected three true positives and missed five, yielding a precision of 1.0, recall of 0.7, and an F1 score of 0.769, showing reliable precision but lower recall, as some instances were not captured.

For hyperparameter tuning, the *LLM-based detectors* showed strong results with five true positives and one false negative, resulting in a precision of 1.0, recall of 0.83, and an F1 score of 0.9, reflecting both high precision and sensitivity.

[3]See, e.g., https://www.trustbit.tech/en/llm-leaderboard-juli-2024 and https://klu.ai/llm-leaderboard

The *heuristic-based detectors* identified four true positives with two false negatives, yielding a precision of 1.0, recall of 0.66, and an F1 score of 0.8, demonstrating good accuracy but a slightly lower ability to detect all instances.

For training and agent configuration, the *LLM-based detectors* achieved five true positives and missed only one, resulting in a precision of 1.0, recall of 0.8, and an F1 score of 0.9, indicating good performance. On the other hand, the *heuristic-based detectors* detected three true positives but missed three, resulting in a precision of 1.0, recall of 0.5, and an F1 score of 0.66, indicating precise detections but limited sensitivity.

2) Open-Source Case Studies Results: Across all case studies (see Table IV) for checkpoint rules, both *heuristic-based detectors* and *LLM-based detectors* showed strong performance in several instances, with high F1 scores indicating a balance between precision and recall. Notably, both detectors achieved perfect F1 scores of 1.0 in case studies such as CS4, CS5, CS6, and CS8 for the LLM-based detectors and CS1 for both detector types, demonstrating robust detection accuracy.

There were instances where performance varied. For example, in CS5, the *heuristic-based detectors* achieved an F1 score of 0.4, which showed they had trouble detecting all relevant instances. Similarly, both types of detectors struggled in CS9, with the *heuristic-based detectors* recording an F1 score of 0.25, while the *LLM-based detectors* performed better at 0.833, although they still fell short of a perfect score. In cases like CS10, where the *LLM-based detectors* reached an F1 score of 0.5, some issues with detection accuracy were noted. Nevertheless, the *LLM-based detectors* generally showed higher accuracy in more complex scenarios, while the *heuristic-based detectors* faced certain limitations.

For hyperparameter tuning (see Table V), the *heuristic-based detectors* showed variable performance across the case studies. For example, in CS2, CS3, CS5, CS6, CS8, and CS10, they achieved F1 scores between 0.667 and 1.0, indicating effective detection in those instances. However, in cases like CS4 and CS7, the heuristic-based detectors struggled, with lower F1 scores of around 0.571 or even 0.0, reflecting their limitations in identifying true positives consistently.

The *LLM-based detectors*, on the other hand, generally demonstrated stronger performance, with perfect F1 scores of 1.0 in CS2 and CS10 and high scores in other instances like CS4 (0.889) and CS9 (0.857), suggesting reliable detection across most cases. However, similar to the heuristic-based detectors, there were a few instances where the LLM-based detectors showed decreased accuracy, such as in CS1, where the F1 score was 0.667.

For training and agent configuration (see Table VI), *heuristic-based detectors* displayed varying levels of effectiveness across the case studies. In some instances, such as CS1, they recorded relatively low F1 scores of 0.286, mainly due to many false negatives. Even when performance improved in cases like CS5 and CS6, the heuristic-based detectors only managed F1 scores of 0.667, which suggests moderate effectiveness.

On the other hand, *LLM-based detectors* demonstrated better overall results, achieving higher F1 scores across several case studies. For instance, CS6, CS7, and CS8 achieved F1 scores of 0.909, 0.923, and 0.923, respectively, showing a good balance between precision and recall. Furthermore, the *LLM-based detectors* recorded an F1 score of 0.857 in both CS4 and CS5, highlighting their effective detection capabilities.

VII. Discussion

This section discusses how the research questions were addressed.

RQ1: The findings across the case studies and the industrial case show that LLM-based detectors are generally better than heuristic-based detectors at checking best practices in RL training pipelines.

For checkpoints (see Table IV), LLM-based detectors achieved high F1 scores in several cases, with perfect scores in CS4, CS5, CS6, and CS8, and an F1 score of 0.933 in the industrial case study. These F1 scores indicate strong accuracy, as they could detect checkpoints with both high precision and recall, even in more complex scenarios. In comparison, while heuristic-based detectors had high precision, they often missed instances, as seen in cases like CS5 and CS9 and the industrial case, where they scored 0.769 due to missed true positives.

For hyperparameter tuning (see Table V), LLM-based detectors consistently performed better, with high F1 scores, including perfect scores in CS2, CS3, and CS10, and an F1 score of 0.9 in the industrial study. Heuristic-based detectors showed more mixed results, with lower F1 scores in some cases and missed true positives in scenarios like CS7, showing their limitations with more complex patterns.

For training and agent configuration (see Table VI), LLM-based detectors again showed higher performance, with strong F1 scores in cases like CS6, CS7, and CS8 and an F1 score of 0.9 in the industrial study, compared to 0.66 for heuristic-based detectors. These values suggest that LLMs can handle more detailed rules in training settings, maintaining good precision and recall.

As explained earlier, we deliberately report the results for GPT-3.5 Turbo in our validation, as it is currently on par with the top non-commercial LLMs. We have also performed tests with the most recent GPT-4o model, which shows further moderate improvements.

Overall, LLM-based detectors proved more effective, flexible, and accurate than heuristic-based detectors for assessing best practices in RL training pipelines. Their ability to handle complex code patterns and varied rules across case studies and industrial contexts makes them a stronger choice for ensuring best practices are met in RL training environments.

While LLM-based detectors have shown significant advantages over heuristic-based detectors in evaluating best practices in RL training pipelines, some notable limitations and considerations must be taken into account. LLMs, especially large models such as GPT-3.5 Turbo and GPT-4o, require significant computational resources, resulting in high operational costs and substantial energy consumption. In many scenarios,

TABLE III
PRECISION, RECALL AND F1 SCORES FOR THE INDUSTRIAL CASE STUDY

Rules on Checkpoints					
	True Positives	False Negatives	Precision	Recall	F1 Score
Heuristic-Based Rule-Based Detectors	3.0	5.0	1.0	0.7	0.769
LLM-Based Rule-Based Detectors	7.0	1.0	1.0	0.8	0.933
Rules on Hyperparameters Tuning					
	True Positives	False Negatives	Precision	Recall	F1 Score
Heuristic-Based Rule-Based Detectors	4.0	2.0	1.0	0.66	0.8
LLM-Based Rule-Based Detectors	5.0	1.0	1.0	0.83	0.9
Rules on Training and Agent Configuration					
	True Positives	False Negatives	Precision	Recall	F1 Score
Heuristic-Based Rule-Based Detectors	3.0	3.0	1.0	0.5	0.66
LLM-Based Rule-Based Detectors	5.0	1.0	1.0	0.8	0.9

TABLE IV
PRECISION, RECALL, AND F1 SCORES FOR HEURISTIC-BASED AND LLM-BASED DETECTORS

Case Study	**Heuristic-based Detectors**			**LLM-based Detectors**		
	Precision	**Recall**	**F1 Score**	**Precision**	**Recall**	**F1 Score**
Rules on Checkpoints						
CS1	1.0	1.0	1.0	0.75	1.0	0.857
CS2	1.0	0.5	0.667	1.0	0.5	0.667
CS3	1.0	0.5	0.667	1.0	0.5	0.667
CS4	1.0	0.5	0.667	1.0	1.0	1.0
CS5	1.0	0.25	0.4	1.0	1.0	1.0
CS6	1.0	0.5	0.667	1.0	1.0	1.0
CS7	1.0	0.5	0.667	1.0	0.5	0.667
CS8	1.0	0.5	0.667	1.0	1.0	1.0
CS9	1.0	0.143	0.25	1.0	0.714	0.833
CS10	1.0	0.667	0.8	1.0	0.333	0.5

TABLE V
PRECISION, RECALL, AND F1 SCORES FOR HEURISTIC-BASED AND LLM-BASED DETECTORS ON HYPERPARAMETER TUNING

Case Study	**Heuristic-based Detectors**			**LLM-based Detectors**		
	Precision	**Recall**	**F1 Score**	**Precision**	**Recall**	**F1 Score**
Rules on Hyperparameter Tuning						
CS1	1.0	0.5	0.667	1.0	0.5	0.667
CS2	1.0	1.0	1.0	1.0	1.0	1.0
CS3	1.0	1.0	1.0	1.0	1.0	1.0
CS4	1.0	0.4	0.571	1.0	0.8	0.889
CS5	1.0	0.5	0.667	1.0	0.667	0.8
CS6	1.0	0.6	0.75	1.0	0.6	0.75
CS7	1.0	0.4	0.571	1.0	0.8	0.889
CS8	1.0	0.5	0.667	1.0	0.667	0.8
CS9	1.0	0.5	0.667	1.0	0.75	0.857
CS10	1.0	0.5	0.667	1.0	1.0	1.0

TABLE VI
PRECISION, RECALL, AND F1 SCORES FOR HEURISTIC-BASED AND LLM-BASED DETECTORS ON TRAINING AND AGENT CONFIGURATION

Case Study	**Heuristic-based Detectors**			**LLM-based Detectors**		
	Precision	**Recall**	**F1 Score**	**Precision**	**Recall**	**F1 Score**
Rules on Training and Agent Configuration						
CS1	1.0	0.167	0.286	1.0	0.5	0.667
CS2	1.0	0.4	0.571	1.0	0.8	0.889
CS3	1.0	0.375	0.545	1.0	0.5	0.667
CS4	1.0	0.5	0.667	1.0	0.75	0.857
CS5	1.0	0.5	0.667	1.0	0.75	0.857
CS6	1.0	0.5	0.667	1.0	0.833	0.909
CS7	1.0	0.571	0.727	1.0	0.857	0.923
CS8	1.0	0.571	0.727	1.0	0.857	0.923
CS9	1.0	0.5	0.667	0.857	0.75	0.8
CS10	1.0	0.6	0.8	1.0	0.833	0.909

heuristic-based detectors may be sufficient, especially for simpler, well-defined tasks. Sometimes, a combination of both approaches makes sense. Our LLM-based detectors are highly modular, with each module designed to check specific occurrences of code fragments. This task would require significant development effort compared to using the LLM directly. Still, this modularity enables precise and predictable interactions with LLMs and ensures that each query is focused and relates directly to a part of a rule.

RQ2: To answer RQ2, we established thirty-one architecture rules in key areas of RL. These areas included, in particular, checkpoints, hyperparameter tuning, and training and agent configuration, including handling single or multi-agent setups.

For checkpoints, we defined ten rules that include aspects such as using a specific directory for storage, setting flexible intervals for saves, adding details about the training environment, and performing checks to ensure data integrity. These help keep track of progress and facilitate the recovery of the training process.

Hyperparameter tuning includes ten rules with an emphasis on defining and organizing parameters clearly, using efficient methods to find the best values, and monitoring how changes impact model performance.

For training and agent configuration, we defined eleven rules with a focus on separating training and tuning parameters, using consistent settings, and tracking each agent's performance to improve repeatability and clarity. These rules also include single versus multi-agent configurations focusing on simpler configurations when possible for single agents and adaptable strategies for multi-agent systems to handle changes and interactions between agents.

VIII. Threats to Validity

In this section, we discuss the potential threats to validity and the steps taken to mitigate these threats.

Internal validity addresses the accuracy of the results and whether they can be attributed to specific interventions rather than other influences. We ensured internal validity by clearly defining and consistently applying rules with our assessment framework. Detection algorithms were carefully implemented and verified to identify true and false positives. All authors cross-checked results to reduce biases, verifying that no RL-specific anti-patterns were introduced by the assessment tool, thus preserving internal validity.

External validity refers to the extent to which our findings can be generalized to other contexts or groups. Our study included a large industrial case and ten open-source case studies covering different RL systems. To further enhance the generalizability of our results, future work will include testing a wider range of RL systems across multiple domains. Although the diversity of case studies in this validation minimizes threats to external validity, expanding the dataset will strengthen the broader applicability of our findings.

Construct validity evaluates whether the study accurately measures the intended concepts. In this work, it is very much linked to the clear specification and accurate detection of RL training best practices. To mitigate potential threats, we based our rule definitions on a thorough review of relevant literature and established practices. Aware that different interpretations and levels of expertise might affect how rules are defined, we refined them by engaging with RL experts from our industrial partner to ensure they truly reflect field standards.

A threat regarding LLM-based detectors is the lack of clarity in some instances, which may lead to their inaccuracy in picking out RL-related practices. Since these detectors are used on broad datasets, they might not always correctly understand specialized RL systems. Their effectiveness depends significantly on the clarity of the prompts; ambiguous prompts can result in missed or incorrect detections. Also, applying the prompts to different projects might produce different results and rule interpretations by the LLMs. Moreover, frequent updates to models can introduce inconsistencies, as newer versions may interpret data differently.

IX. Conclusions and Future Work

This work introduces a method for evaluating best practices in RL training using LLMs and heuristic-based detectors. Key areas of focus include checkpoints, hyperparameter tuning, and agent configuration. We applied this approach across various RL systems, covering both industrial applications and open-source projects.

Our findings indicate that LLM-based detectors generally provide better results than heuristic-based ones, especially when working with more complex code patterns. In multiple case studies, the LLM approach achieved greater precision and recall, successfully identifying best practices within RL training pipelines. The modular design of our framework, with many supporting components and rather small modular detectors, is essential for ensuring precision and traceability in the detection process. This modularity is also advantageous in settings with multiple RL projects and CI/CD pipelines, enabling ongoing monitoring and improvements. It is important to consider the limitations associated with LLM-based detectors. These include the high computational cost and energy consumption. Furthermore, while LLMs offer superior performance in complex scenarios, heuristic detectors may still be sufficient for simpler tasks and provide a cost-effective and efficient alternative that can also be used in combination with LLM-based detectors. Future work will include more diverse and complex case studies to further validate the framework. Future work will build on this approach to accommodate more diverse and complex case studies to further validate the framework. We will refine current rules and create new ones tailored to different algorithms and frameworks. Furthermore, we plan to incorporate this rule-based evaluation into continuous integration and deployment pipelines, allowing for real-time validation of RL practices to enhance the reliability and efficiency of these systems.

X. Acknowledgments

This work was supported by the FFG (Austrian Research Promotion Agency) project MODIS (no. FO999895431).

REFERENCES

[1] Canese, L., Cardarilli, G.C., Di Nunzio, L., Fazzolari, R., Giardino, D., Re, M., Spanò, S.: Multi-agent reinforcement learning: A review of challenges and applications. Applied Sciences **11**(11), 4948 (2021)

[2] Chen, H., Lundberg, S., Lee, S.I.: Checkpoint ensembles: Ensemble methods from a single training process. arXiv preprint arXiv:1710.03282 (2017)

[3] Eimer T., Lindauer M., R.R.: Hyperparameters in reinforcement learning and how to tune them. In: International Conference on Machine Learning. pp. 9104–9149 (2023)

[4] Eisenman, A., Matam, K.K., Ingram, S., Mudigere, D., Krishnamoorthi, R., Annavaram, M., Nair, K., Smelyanskiy, M.: Check-n-run: A checkpointing system for training recommendation models. arXiv preprint arXiv:2010.08679 **5** (2020)

[5] Hao, J., Yang, T., Tang, H., Bai, C., Liu, J., Meng, Z., Liu, P., Wang, Z.: Exploration in deep reinforcement learning: From single-agent to multiagent domain. IEEE Transactions on Neural Networks and Learning Systems (2023)

[6] Hernandez-Leal, P., Kartal, B., Taylor, M.E.: A survey and critique of multiagent deep reinforcement learning. Autonomous Agents and Multi-Agent Systems **33**(6), 750–797 (2019)

[7] Lakshmanan, V., Robinson, S., Munn, M.: Machine Learning Design Patterns. O'Reilly Media, Inc. (October 2020)

[8] Lee, D., He, N., Kamalaruban, P., Cevher, V.: Optimization for reinforcement learning: From a single agent to cooperative agents. IEEE Signal Processing Magazine **37**(3), 123–135 (2020)

[9] Li, L., Jamieson, K., DeSalvo, G., Rostamizadeh, A., Talwalkar, A.: Hyperband: A novel bandit-based approach to hyperparameter optimization. Journal of Machine Learning Research **18**(185), 1–52 (2018)

[10] Mnih, V., Kavukcuoglu, K., Silver, D., et al.: Human-level control through deep reinforcement learning. Nature **518**(7540), 529–533 (2015)

[11] Samsami, M.R., Alimadad, H.: Distributed deep reinforcement learning: An overview. arXiv preprint arXiv:2011.11012 (2020)

[12] Schneider, S., Quéval, P.J., Milánkovich, Á., Díaz Ferreyra, N.E., Zdun, U., Scandariato, R.: Automatic rule checking for microservices: Supporting security analysis with explainability. Available at SSRN 4658575 (2023)

[13] Schulman, J., Wolski, F., Dhariwal, P., Radford, A., Klimov, O.: Proximal policy optimization algorithms. arXiv preprint arXiv:1707.06347 (2017)

[14] Schwartz, H.M.: Multi-agent machine learning: A reinforcement approach. John Wiley & Sons (2014)

[15] Sharma, R., Davuluri, K.: Design patterns for machine learning applications. In: 2019 3rd International Conference on Computing Methodologies and Communication (ICCMC). pp. 818–821 (2019)

[16] Stol, K.J., Ralph, P., Fitzgerald, B.: Grounded theory in software engineering research: a critical review and guidelines. In: Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on. pp. 120–131. IEEE (2016)

[17] Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction. MIT Press (2018)

[18] Washizaki, H., Khomh, F., Guéhéneuc, Y.G., Takeuchi, H., Natori, N., Doi, T., Okuda, S.: Software-engineering design patterns for machine learning applications. Computer **55**(3), 30–39 (2022)

[19] Zhang, K., Yang, Z., Başar, T.: Multi-agent reinforcement learning: A selective overview of theories and algorithms. Handbook of reinforcement learning and control pp. 321–384 (2021)

[20] Zhang, K., Yang, Z., Başar, T.: Multi-agent reinforcement learning: A selective overview of theories and algorithms. Handbook of reinforcement learning and control pp. 321–384 (2021)