universität wien

MASTERARBEIT | MASTER'S THESIS

Titel | Title

Pipeline Quality Insight as a Service for Pattern Detection and Quality Assessment in MLOps Systems

verfasst von | submitted by

Dott. Francesco Urdih

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

Master of Science (MSc)

Wien | Vienna, 2024

Studienkennzahl lt. Studienblatt | Degree programme code as it appears on the student record sheet:	UA 066 921
Studienrichtung lt. Studienblatt | Degree programme as it appears on the student record sheet:	Masterstudium Informatik
Betreut von | Supervisor:	Univ.-Prof. Dr. Uwe Zdun

Acknowledgements

First, I would like to express my gratitude to my supervisor, Professor Uwe Zdun, for the trust put in me. Then, this journey would not have been possible without Evangelos Ntentos and his valuable feedback and experience.
Many thanks to my family and girlfriend for their moral support, which kept my morale high during this process. Lastly, thanks to my friends for always standing by.

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

Abstract

The growing interest in Machine Learning (ML) applications has increased the significance of the MLOps (ML Operations) field, which aims to bring ML to production. At the same time, Reinforcement Learning (RL) popularity has risen in artificial intelligence, allowing the training and usage of intelligent agents. For both areas, collections of principles and techniques to apply have already been proposed. Nevertheless, given the complexity of MLOps and RL systems, identifying these patterns and practices from source code is time-consuming. We investigate 9 categories of patterns and practices and 7 technologies to apply them. We propose a Pipeline Quality as a Service approach, to automatically extract insights, related to these 9 categories, directly from source code. We evaluate our approach using 15 case studies. Our framework reaches a high level of accuracy with an F1-score of 0.99 in detecting MLOps-related patterns and practices and 0.82 for the RL-related ones. These results highlight the approach's applicability for automatic MLOps and RL architectures analyses.

Kurzfassung

Das wachsende Interesse an Machine Learning (ML) Anwendungen hat die Bedeutung des Bereichs MLOps (ML Operations) erhöht, dessen Ziel es ist, ML in die Produktion zu bringen. Gleichzeitig hat das Reinforcement Learning (RL) in der künstlichen Intelligenz an Popularität gewonnen und ermöglicht das Trainieren und die Nutzung intelligenter Agenten. Für beide Bereiche wurden bereits eine Reihe von Prinzipien und Techniken vorgeschlagen, die angewendet werden können. Angesichts der Komplexität von MLOps und RL-Systemen ist es jedoch zeitaufwändig, diese Muster und Praktiken im Quellcode zu identifizieren. Wir untersuchen 9 Kategorien von Mustern und Praktiken und 7 Technologien, um sie anzuwenden. Wir schlagen einen Pipeline Quality as a Service-Ansatz vor, um automatisch Erkenntnisse zu diesen 9 Kategorien direkt aus dem Quellcode zu gewinnen. Wir evaluieren unseren Service anhand von 15 Fallstudien. Unser Framework erreicht eine hohe Genauigkeit mit einem F1-Wert von 0,99 bei der Erkennung von MLOps-bezogenen Mustern und Praktiken und 0,82 für RL-bezogene Muster. Diese Ergebnisse unterstreichen die Anwendbarkeit des Ansatzes für automatische Analysen von MLOps- und RL-Architekturen.

Contents

List of Tables

List of Figures

Listings

1. Introduction

In this thesis, we analyse 9 categories of patterns and practices, 7 for RL and 2 for MLOps, providing their impact on a system. Then, we propose a framework to detect such patterns and practices relying exclusively on a project's source code.
This chapter motivates the need for this work, formulates the questions it intends to address, and describes the thesis format.

1.1. Motivation

Systems in which ML (Machine Learning) models play an essential part are drawing more interest from practitioners [52]. In these systems, *classical* software engineering and DevOps practices are not enough as they do not tackle the specific needs of ML. These and other challenges (e.g. business-related) lead many ML-related projects to fail [46]. To help reduce technological debts and limit the possibility of failure, the MLOps field is gaining significance [45].
Similarly to MLOps, the Reinforcement Learning (RL) field is gaining interest among practitioners in training models able to make complex decisions inside live environments.

Both research areas present a vast list of choices for principles, technologies, patterns, and best practices to be applied [66, 54, 62, 26]. Nevertheless, we notice a lack of tools assisting in the detection of the most common patterns and practices. For these reasons, it is not easy for practitioners to get to know a new system they encounter. Investigating which patterns and practices are used and which are not requires time and is error-prone. When systems are on a large scale, the manual effort required is not realistic.
An automatic identification of the most common MLOps and RL patterns and practices directly from the source code would better support these analysis processes.

1.2. Problem Statement

To approach the topic's broadness and complexity, we must first define our problem statement. This is distilled into 3 research questions:

- **RQ1**: Which patterns and practices are currently used by practitioners in MLOps and RL-based architectures?

- **RQ2**: How accurately does the detection mechanism ensure compliance with patterns and practices identified in RQ1?
- **RQ3**: How well can the patterns and practices identified in RQ1 be automatically detected from a system's source code?

More in detail, RQ2 focuses on the performance reached by our proposed framework during the detection process, while RQ3 is related to its level of automation.

1.3. Structure of the Thesis

The work is structured as follows:

- The second chapter contains the foundation to understand the work and to achieve a homogeneous comprehension of concepts inside the thesis. Basic concepts such as *MLOps* and *RL* are presented and explained there.
- The third chapter presents the methodology applied in this thesis.
- A comprehensive review of all the patterns, practices and technologies detected in our framework is provided in the fourth chapter. RL patterns and practices are presented in the first section, MLOps ones are presented in the second section, and technologies are presented in the third.
- In the fifth chapter, all the information related to our framework implementation can be found. Besides describing the main components of our framework, we also define all the source code elements we are interested in.
- In the sixth chapter, we describe the evaluation process for the proposed framework and the results obtained using 15 RL and MLOps case studies. In addition, we briefly discuss how we integrated our application into an existing codebase internally used by a big software company.
- We discuss the main findings of this thesis in the seventh chapter, answering each research question. Furthermore, recommendations for future research are given here.
- In the eighth chapter, we present related work, focusing on the novice of this thesis and what this thesis does similarly and differently from other works.
- In the ninth chapter, we discuss the validity threats of this work and provide suggestions to address these threats in future work.
- The tenth chapter concludes this master thesis, summarising the key results.

2. Fundamentals

Understanding a few key concepts is fundamental to comprehend the topic of this thesis. This chapter aims to introduce such concepts before exploring the details of this work.

2.1. Reinforcement Learning

Reinforcement Learning, usually shortened in RL, is a branch of machine learning that shifts the paradigm from learning and using patterns in the data to learning using feedback. Sutton et al. [61] define Reinforcement Learning as:

> Learning what to do-how to map situations to actions-so as to maximize a numerical reward signal. The learner is not told which actions to take, but instead must discover which actions yield the most reward by trying them.
> Actions may affect not only the immediate reward but also [...] all subsequent rewards. These two characteristics—trial-and-error search and delayed reward—are the two most important distinguishing features of reinforcement learning.

RL is radically different from *traditional* ML (i.e. supervised and unsupervised learning) and aims to solve completely different kinds of problems. The goal is to train an *agent* (i.e. a program) to make decisions in an environment based on the state of the environment. The applications for RL are numerous; among others, we can find examples in the literature where researchers taught an agent to play Atari games [49] such as Pong and Space Invaders, schedule read and write requests on DRAM [32], recommend news to read [69], regulate temperature and airflow for server rooms in data centres [39].
The branch lays its foundation on research conducted decades ago; nevertheless, in the last 10 years, it has seen rapid progress, especially in its techniques (i.e. algorithms) and applications. The patterns, practices and technologies that emerged up to this point will be analysed in Chapter 4.

It's essential to formalise its fundamental principles to understand Reinforcement Learning properly. Typically in an RL setting, we find an *agent* and an *environment*. At every time step t, the agent has to choose an action a based on the current state s_t.

Actions can be picked from a predefined set A, which can be discrete or contiguous, depending on the environment. The action a is chosen based on the agent's policy π, which maps states to probability distributions over all the actions [61]. After selecting the action a_t, the environment will return a reward r_t and a new state s_{t+1} based on the respective probability distributions. A simplified view of such interactions can be seen in Figure 2.1.
Training in Reinforcement Learning typically means finding the *best* possible policy π. More formally, to compare policies, we define the *discounted return*:

$$G_t = \sum_{k=0}^{+\infty} \gamma^k r_{t+k+1} \tag{2.1}$$

which is the sum of *discounted* rewards received in future time steps. γ is the *discount rate*, $0 \leq \gamma \leq 1$ and it is used to increase the weight (i.e. importance) of rewards closer in time over later ones. Using the discounted return, we define the *value function*:

$$v_\pi(s) = \mathbb{E}_\pi[G_t | S_t = s] \tag{2.2}$$

which is the expected return under the policy π, starting from the state s. Using $v_\pi(s)$ we can say that a policy π' is better than a second policy π'' if:

$$v_{\pi'}(s) > v_{\pi''}(s) \quad \forall s \in S \tag{2.3}$$

where S is the set of all possible states.

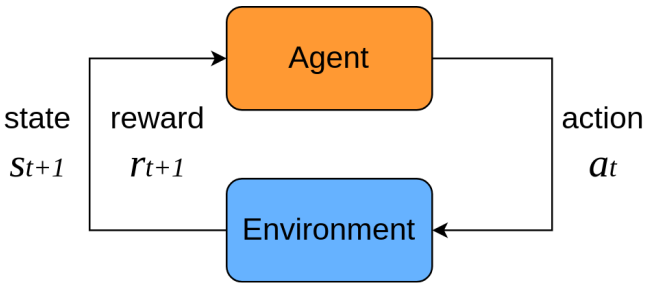

Figure 2.1.: The loop in a typical reinforcement learning scenario, containing one environment and one agent

2.2. Workflows

Workflows are any repeatable sequences of actions. They are related to any kind of work, but we will refer only to computer science and specifically to *human-to-system processes* in process-driven applications.

In this regard, workflows are defined as configurable and mostly automated processes that run defined activities. They can be imagined as a blueprint of operations. Common activities for our scenarios are: building software, testing it, and deploying it.
Usually, workflows group actions together if they are deeply connected. These groups are called `jobs` for the technologies we analyse. Many technologies exist to implement workflows; two will be discussed in Section 4.3.

2.3. DevOps

DevOps [17] is an expression obtained from merging two words: development and operations. Specifically, these refer to *software development* and *IT operations*. It is one of the foundations of MLOps, therefore to understand MLOps we must begin with DevOps.
DevOps is ultimately an umbrella term for software engineering principles that combine multiple phases of IT projects: development, deployment and operations. In the past (up to the late 2000s), it was common practice for any organization to have *barriers* between who developed software and who made it operational. This resulted in highly dysfunctional, inefficient and slow processes which DevOps intends to improve. To do so, a close collaboration should exist between the development and the operations phases, with a philosophy of *shared responsibility* and *shared ownership*.
In practical terms, DevOps principles push towards the (partial) automation of the software development lifecycle, particularly testing, configuring environments and deploying.
Faustino et al. [20] detail the main benefits of DevOps, linking them to specific case studies. These benefits comprehend:

- an increase in code quality
- an increase in customer satisfaction
- a reduction in costs
- a reduction in time to market

2.4. MLOps

MLOps, like DevOps, is an expression coined combining ML (i.e. *Machine Learning*) and operations (i.e. *IT Operations*). It is an *extension* to DevOps, specific to applications which include ML models. The latter have additional requirements, compared to *classical* non-ML systems, and present radically different types of issues. These issues mostly come from two key reasons:

- **the presence of training data**. Version control systems (e.g. Git), traditionally

designed to track source files, lose efficacy (i.e. the ability to track) and efficiency (i.e. the space used to track) when tracking datasets. Additional practices are needed besides the ones inherited from DevOps.

- **the volatile and unpredictable performances of live ML models**, caused by changes between training data and live input data. This is often referred to as *concept drift* [44]. This behaviour requires more monitoring than non-ML systems and more organized versioning of deployed ML models.

Both problems (and others) are present only in applications running ML models, therefore they are not addressed by DevOps techniques, designed considering *classical* IT projects' requirements. This lack forces an evolution of DevOps.

Kreuzberger et al. [36] define MLOps as:

> sets of concepts [...] when it comes to the end-to-end conceptualization, implementation, monitoring, deployment, and scalability of machine learning products.

Since MLOps is a progression of DevOps, many DevOps patterns and practices are applied in the former but are often extended to include ML in them. These will be thoroughly analysed in Chapter 4.

2.5. Software Patterns and Practices Detection

Significant relevance in software development is given to patterns and best practices. Understanding which pattern or practice is applied in a system is denoted as *patterns and practices recognition*. Typically, a practitioner would manually explore a system's documentation and source code whenever he needs to know whether this system applies one or more specific patterns and practices. For big codebases, the effort required by this analysis is considerable. Therefore, an essential area in software engineering research is assisting a practitioner in completing the analysis process. Based on the specific patterns and practices a practitioner may be interested in, there can be tools to help find them. The nature of these tools can significantly vary in how they can be used (e.g. a CLI, a GUI, an IDE extension) and the amount of help they give. For example, Fontana et al. [22] proposed an Eclipse[1] plugin, which detects design patterns from Java source code.

[1] `https://eclipseide.org/`: accessed September 2024

3. Methodology

In this Chapter, we discuss the methodology followed in this thesis, outlined in Figure 3.1. In the following paragraphs, we describe the 6 steps we have pursued, dividing them into 3 phases: *Research*, *Implementation* and *Evaluation*.

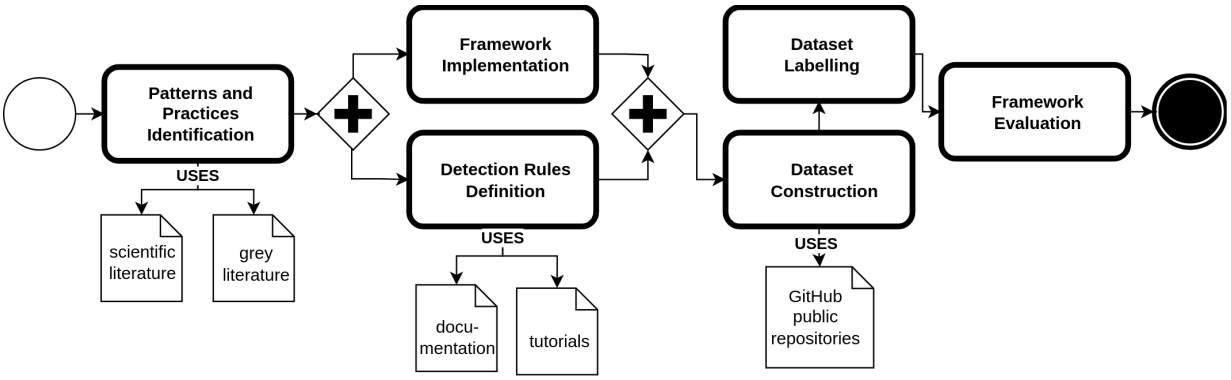

Figure 3.1.: Overview of the methodology used in this thesis.

Research

The research phase focuses on answering RQ1. In the *Patterns and Practices Identification* step, we have reviewed both scientific and grey literature[1] (see Table A.1) to identify RL and MLOps patterns and best practices. Based on this analysis, we have defined 9 categories of patterns and practices, presented in detail in Chapter 4.

Implementation

After having answered RQ1, we moved to the implementation phase, presented in Chapter 5. At this stage, we have implemented a framework composed of three different modules:

- A Java application to analyse workflow files.
- A Python application to analyse Python files.
- An Angular application to display the analysis results.

Sections 5.2, 5.3, and 5.4 illustrate in detail this step.

[1] Grey literature is composed of practitioner books, videos, blogs, online presentations, etc.

In parallel with the development, we have defined the rules used by the framework to find patterns and practices from source code. To craft these rules, we have relied upon tutorials and technical documentation of the technologies we analyse, presented in Section 4.3. More details for this step can be found in Section 5.1.

Evaluation

The final phase in our methodology is the evaluation, described in Chapter 6, for which we have followed three steps. First, we have constructed a dataset of 15 MLOps and RL case studies, mostly relying on open-source repositories on GitHub. Then, we manually labelled this dataset to identify where, in the source code, each investigated pattern and practice was applied. Finally, we have evaluated the results produced by our framework, using the labelled dataset to process the evaluation metrics.

4. Background

This chapter describes the 9 categories of patterns and practices we investigated in this thesis. We use scientific studies and grey literature from practitioners (i.e., blogs, videos, and presentations) for our arguments. The practitioners' sources are listed in the appendix in Table A.1.
We conclude the chapter with an overview of specific technologies to apply the investigated categories.

4.1. Reinforcement Learning Patterns and Practices

In this section, we review 7 categories of patterns and practices, highlighting their importance for practitioners.

4.1.1. Checkpoints

It's common while training RL models to run into errors, e.g. bugs in the user's program or hardware failures, which may cause data loss (i.e. training progress) and time (i.e. all the episodes run before the errors occurred). Checkpoints [38] limit the damage by periodically saving the current state of RL models, usually on disk. Using them, the training can resume from the last saved state whenever a bug arises instead of restarting from the beginning. Their recovery function greatly improves the efficiency of the training process, especially for complex RL models using deep neural networks, where around 20% of jobs can fail before completion [33].
Their usage comes with two main costs: storage and computation overhead. The former results from saving models on disk, especially when hundreds to thousands of checkpoints are to be saved. The latter is caused by additional logic when performing a checkpoint since the training process is temporarily halted. Nevertheless, for this practice, the benefits largely surpass the costs.
Furthermore, checkpoints are key when using the *early stopping* strategy [S12]. This consists of periodically assessing a model's performance and stopping the training if performances don't improve for a defined number of epochs. Early stopping uses the saved checkpoints to load back the model with the best performances.

Note that there are two phases of the checkpoint practice: saving and loading[1] but

[1] Loading refers to reading a checkpoint data from disk and creating an RL model from it.

we're interested in detecting only the first.

Besides the checkpoint best practice, there are two other related patterns: checkpoint path and checkpoint frequency. The checkpoint path is the location where one or more checkpoints are saved. Many libraries allow not to specify it and instead use a default location, which is not ideal.
The checkpoint frequency represents how often checkpoints are performed, and it's as important as the practice itself. Too often and too rarely may both lead to considerable inefficiency. The first causes high computation overheads, and the second leads to many computations being lost in case of errors. It's, therefore, key for practitioners to know whether these two patterns are applied in a project.

4.1.2. Hyperparameter Tuning

When training models in Reinforcement Learning, there are dozens of hyperparameters [5] which can be set. Depending on the specific one, different values can change the model's performance, training efficiency, computation cost, and many other metrics [18, 5]. For these reasons, it's important to set most hyperparameters properly. Although studies suggest default values for many parameters [5], it's impossible to find values that work for every scenario. For this reason, during the training phase, it's important to train multiple times an RL model with different values and choose the *best* ones. This practice is referred to as *Hyperparameter Tuning*, and it's not unique to Reinforcement Learning but already applied to Machine Learning.
Tuning hyperparameters means training the RL model for each vector of values, which in turn means consuming computation resources for each chosen vector. Methods have been developed to choose new vectors to test and reduce the resources spent. Awad et al. [7] propose an evolutionary algorithm to generate new hyperparameter vectors using the previously suggested ones and the result performance metrics.
In this thesis, we are not interested in how the tuning practice is applied but whether it is.

4.1.3. Distributed Reinforcement Learning

In Reinforcement Learning, like most computer science branches, one common problem is how to speed up the time required for computations. One technique to reduce such time is distributing the learning process among multiple workers and running multiple RL environments simultaneously. This pattern is referred as *distributed RL* [59].

As already mentioned in Section 2.1, an RL algorithm learns an agent by sampling rewards from an environment. The speed of training an RL agent depends, among other things, on the number of actions per unit of time which can be taken in the training environment. This factor can be increased by running multiple instances of an environment in parallel, potentially reducing the learning time[2]. Running multiple instances of an environment only requires a bigger memory size and (ideally) numerous CPU cores or one or more GPUs to be used. Both requirements can be easily satisfied with the cheap cost of modern hardware. Nevertheless, using parallel environments comes with a new challenge: designing an algorithm capable of simultaneously learning from multiple running environments. Parallel and distributed algorithms are challenging in computer science, and Reinforcement Learning is no exception.
Such an RL algorithm must be able to update the policy π with parallel feedback, and this requirement cannot be satisfied for many RL state-of-the-art algorithms (e.g., Value Iteration [61], Policy Iteration [61]), designed before *distributed training* was a desired property.
Besides speeding up training, another advantage of distributed RL is more *robustness* of the algorithm [48], if this uses independent sets of parameters for each worker. Using more than one set allows more exploration of the environment and more likelihood for the algorithm to learn a close-to-optimal policy π.

To understand how distributed Reinforcement Learning works, we can analyse the A3C (Asynchronous Advantage Actor-Critic) algorithm [48], illustrated in Figure 4.1. A3C is designed to manage multiple workers, each using its own parameters and learning using a separate instance of the environment. These workers collect experiences from their environment and independently update the policies, processing the learning gradients. These adjustments are then sent to the *global networks*, which update asynchronously the policy $\pi(s)$ and the value function $V(s)$. At the end of the training phase, the global $\pi(s)$ and $V(s)$ is the learned model. Reinforcement Learning state-of-the-art algorithms comprehend many other designs [30, 53, 49, 19] and A3C design is just an example. Other architectures may differ regarding the number of networks, whether they are asynchronous or use replay buffers, etc.

It's imperative to note that the distributed nature of an algorithm is not necessarily intrinsic to the algorithm but to the implementation. While it's true that some algorithms are already designed in a distributed fashion, there are cases where an algorithm was initially proposed without allowing distributed learning. However, given its success, it has been modified to learn from multiple environments simultaneously. An example is the Proximal Policy Optimisation (PPO) [60] algorithm, so

[2] Using n environments does not automatically translate in $1/n$ training time because distributed RL has worse sample efficiency.

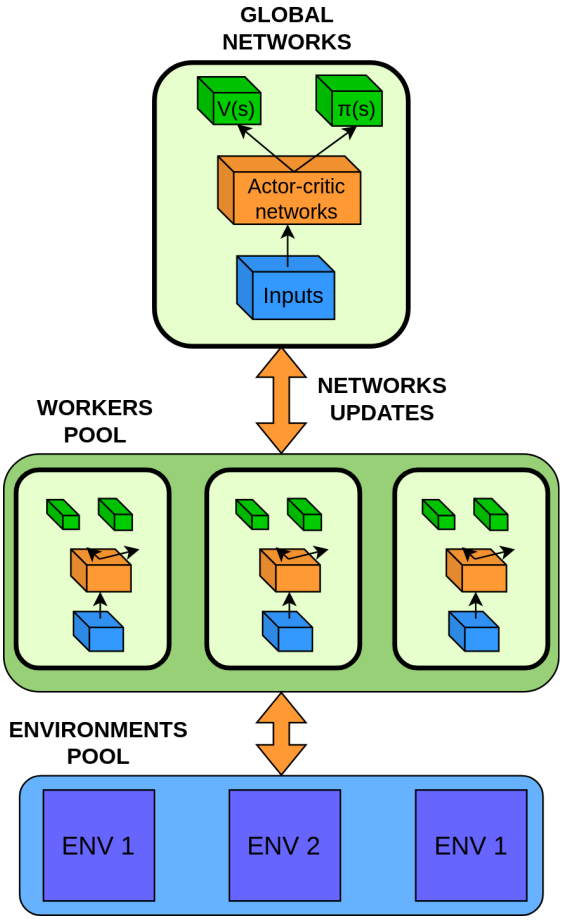

Figure 4.1.: Architecture of the A3C algorithm. Multiple workers, with their networks for policies and values functions, learn independently from different environment instances. Updates are then sent asynchronously to the global network. Image adapted from [2].

popular among practitioners that it was proposed in a distributed fashion, in multiple versions: Asynchronous Proximal Policy Optimisation (APPO) [1] and Distributed Proximal Policy Optimisation (DPPO) [28].
Ultimately, when investigating whether a codebase is applying distributed RL, it's not enough to know the algorithms utilised. Knowledge about the specific implementation (i.e., the library used) is necessary.

4.1.4. Single/Multi-Agent Reinforcement Learning

In Section 2.1 we have mentioned a few applications of reinforcement learning. All of these assume the presence of a single controller, i.e. an RL agent, to work. Never-

theless, it's easy to think of other scenarios where we can find multiple independent controllers, possibly with different goals, all interacting in the same environment. This practice is denoted as *Multi-Agent Reinforcement Learning*, or *MARL*. When MARL is not applied, and we find only one agent in one environment, we denote the practice as *Single-Agent Reinforcement Learning*, or *SARL*. We are interested in identifying both patterns. Nevertheless, in this section, we focus on MARL as the more common Single-Agent scenario has already been introduced in Section 2.1.

A simplified view of MARL is shown in Figure 4.2. There, we can see that not only do agents choose different actions $a_{i,t}$ but also receive different observations $s_{i,t+1}$ and rewards $r_{i,t+1}$[3].

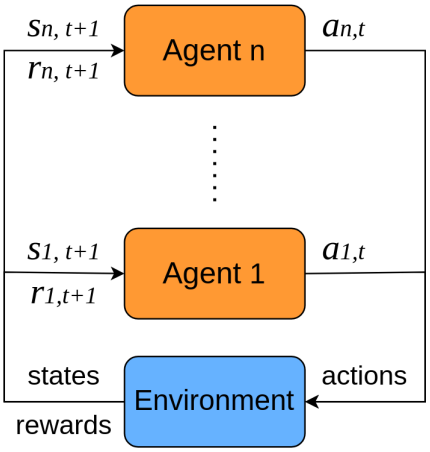

Figure 4.2.: Architecture of Multi-Agent Reinforcement Learning. The loop contains one environment and N agents.

MARL is not so recent from a theoretical perspective [11], but its usage falls behind compared to the simpler SARL counterpart. Therefore, there are also fewer technologies to apply the practice, as shown in Section 4.3. Nevertheless, in recent years, more and more scenarios have been investigated. Yang et al. [67] trained multiple agents to control one UAV (Unmanned Aerial Vehicles). Terry et al. [63] developed a series of MARL environments, among which we can find 1v1 games such as *tennis* and *pong* where two agents compete with each other. Baker et al. [8] taught four agents to play hide and seek, where the agents were split into teams of two each.

[3] Different observations are used for a partial view of the environment state, while different rewards are used for different goals. Note that these characteristics depend on the specific environment and are not applied in every MARL setting.

In certain scenarios, like the one studied by Yang et al., MARL is necessary to overcome the high computational complexity. In their case, multiple UAVs have to be controlled, and their solution is to have one agent per vehicle. Nonetheless, since all the vehicles have the same shared goal, it could be conceived to use one central agent to control all UAVs simultaneously. Although possible, this solution leads to an intractable problem because of the size of the action set A^4.

These applications highlight one fundamental characteristic of MARL environments: the relationship between agents. Agents can have the same goal and cooperate like in the example of Yang et al., a pattern denoted as *fully cooperative* MARL [55, 68]. Or they can all have different colliding goals, like for Terry et al. [63], a pattern called *fully competitive* MARL [68]. A third option is a mixed setting of the two, where agents both compete and cooperate. This pattern is denoted as *mixed cooperative-competitive* MARL [68], and the study of Baker et al. is an example of application. Distinguishing the kind of MARL in place is crucial not only to get a more precise overview of the application but also because it heavily affects other design decisions:

- The *algorithm*. Many algorithms are designed to work only in one scenario, while others can be used in multiple ones. Additionally, some models may be optimised for one MARL setting and not for others. Knowing the type of relationship between agents helps practitioners select the algorithms to be employed.
- The *environment architecture*. Different scenarios can drastically change the architecture of the environment. For example, in fully cooperative MARL, communication between agents can improve performance [34]. This ultimately requires the environment to offer communication channels.

4.1.5. Reinforcement Learning Pipeline

When writing a program with Reinforcement Learning, there are some standard stages to define. Using the RL-related sources from table A.1, we have tried to identify some critical steps of the RL pipeline, not tied to a specific library:

- Definition[5] of a new environment.
- Creation of an environment.
- Creation of an RL agent.
- Creation of an RL agent.
- Training an RL agent.
- Saving an RL agent.
- Evaluating an RL agent.

[4] A is the result of all the possible combinations of a single agent's actions.
[5] By *definition*, we mean writing the logic of the environment, encapsulated in a class.

Some of these steps are always present (e.g. environment and agent creation), while others may be missing but are relevant for the detection (e.g. agent saving). Furthermore, specific steps are a best practice (e.g. agent saving, agent evaluation). The rest can be considered patterns. We have chosen to include all these patterns and best practices under one category.

It's important to note that the order of the steps is flexible. Some steps may be before or after others, like the initialisation of the environment and the algorithm. Other steps follow a strict order: saving and evaluating a model must be executed after training such a model.

Inside the *agent training* pattern, we have chosen to include the best practice of Continuous Training (CT) [15], which consists of periodically training an agent. Even though this practice has originated as MLOps best practice, we have chosen to include it inside this RL category.

4.1.6. Training Configuration

In the previous section, we have presented the most important steps of a Reinforcement Learning program. Now, we are interested in analysing two key characteristics of the training phase:

- Hyperparameters usage, in particular the training duration, the discount rate and the learning rate.
- Seeding.

We discuss the configurations separately, highlighting their importance.

Hyperparameters Usage

We have already introduced the hyperparameter tuning practice and explained its relevance in finding the *best* RL model. Besides knowing whether a tuning phase is used, practitioners must understand which hyperparameters are defined in the project and which are left as library defaults.
Given the high number of hyperparameters that we can find in Reinforcement Learning [5], both universal (i.e. for any algorithm) and algorithm-specific, we focus on three of the most impactful ones: training duration, discount factor, and learning rate.

The duration of the training phase is the most important hyperparameter in RL as this determines whether the algorithm has enough experience to reach a quasi-optimal policy. There are three ways to express the training duration:

- Number of episodes. An episode is the sequence of tuples s, a, r (state, action, reward) that occur from the starting state of an environment to its terminal

state.
- Number of steps. A step is one gradient update for the trained model.
- Number of epochs. An epoch is a complete iteration over the training set.

The *discount rate* γ is another key hyperparameter that should be set for every model. It weighs the importance of newer rewards compared to older ones and deeply affects the performance (e.g., average reward) of trained agents [5].

The third and last hyperparameter we want to detect is the *learning rate*, which weights the gradient updates over the learning function. It is generally important to set, [24] p. 429, because when it's too small it may increase the training period, when it's too big it may lead to suboptimal solutions.

Seeding

To understand the performance of a trained agent, multiple episodes in one environment have to be executed. Similarly, multiple episodes must be run for each tested vector when applying hyperparameter tuning. Because of the non-determinism of most RL environments[6], these two tasks become harder. Ideally, we would want to remove the non-determinism. This is done by setting randomness seeds that operate on the pseudo-random functions inside RL environments and agents. Seeding is an important configuration for any practitioner, making experiments repeatable and limiting random chance from affecting the evaluation phase [S9][S10].
Furthermore, seeds can make the training phase more robust when using multiple seeds because these allow running an agent in more than one scenario [S10][S13]. Note that, for some libraries, seeds can also be used for agents to get deterministic results.

4.2. MLOps Patterns and Practices

This section reviews 2 MLOps categories of patterns and practices, highlighting their importance for practitioners.

4.2.1. Workflow Automation

Practitioners need to automate processes inside a codebase. The goals of automation are very broad; they can span from a simple build process every day at a particular hour to tests and deployments at every `git push`, without the need for human intervention. Regardless of which process is automated, the benefits are many, among which we can find error reduction, cost reduction, and team empowerment

[6] Given a tuple (s_t, a_t), the next state and reward (s_{t+1}, r_{t+1}) are not deterministic

[21, 31, 16]. In this thesis, we want to detect whether process automation is present in a project. Specifically, we look for workflows. Besides using workflows, we want to extract key information relevant to practitioners, such as the structure (i.e. the jobs present) and the jobs' dependencies.

4.2.2. Workflow Results

While running a workflow, many commands are executed. Only a part of these commands produces results that are useful to other jobs or a practitioner investigating how the workflow execution has proceeded. In this thesis, we are interested in extracting and displaying part of said results in our framework.
More specifically, given the great number of outputs a job or a workflow yields, we are interested in two types of results:

- Job artifacts[7][8]. These can include, among many others, build results to use for testing and deploying[9].
- Container images.

4.3. Technologies

This section reviews seven popular [4] critical technologies used to apply the patterns and practices presented in the previous sections.

4.3.1. RLlib

RLlib[10] [41], `v2.34.0` at the time of writing, is a very popular open-source reinforcement learning Python library which implements several state-of-the-art RL algorithms[11] focusing on their parallelisation. To implement such algorithms, the library relies on `ray` [50] parallel abstractions.
Its goal is not only to offer RL algorithms out-of-the-box with a common API but also to seek more standardized implementations of RL models by using said abstractions. The library has a codebase of considerable size[12] and, as we show in Section 5.1, this

[7] `https://docs.gitlab.com/ee/ci/jobs/job_artifacts.html`: accessed September 2024
[8] `https://docs.github.com/en/actions/writing-workflows/choosing-what-your-workflow-does/storing-workflow-data-as-artifacts`: accessed September 2024
[9] `https://docs.github.com/en/actions/writing-workflows/choosing-what-your-workflow-does/storing-workflow-data-as-artifacts#uploading-build-and-test-artifacts`: accessed September 2024
[10] `https://docs.ray.io/en/releases-2.34.0/rllib/index.html`: accessed September 2024
[11] `https://docs.ray.io/en/releases-2.34.0/rllib/rllib-algorithms.html`: accessed September 2024
[12] `https://github.com/ray-project/ray/tree/master/rllib`: accessed September 2024

results in multiple ways to apply the practices presented in Section 4.1.

Note that even though RLlib is the specific `ray` component for Reinforcement Learning, other elements are often used to apply RL practices, as we show in Section 5.1. Furthermore, RLlib has a contribution package (`rllib-contrib`[13]), which gathers experimental implementations of RL algorithms. We did not include this package in this thesis for these reasons:

1. it is rarely used, compared to the official package
2. the implementations may never be published
3. the implementations are experimental. Therefore, the APIs may easily change and require updates on the detection mechanisms

4.3.2. Stable-Baselines3

Stable-baselines3[14], `v2.3.2` as of September 2024, [57], often referred to as SB3, is another popular RL Python library which implements state-of-the-art reinforcement learning algorithms. Its primary focus is offering simple API to improve usability. Therefore, contrary to RLlib, there are fewer ways (i.e. functions, classes, etc.) to apply practices. Like RLlib, SB3 has a contribution package (`stable-baselines3-contrib`[15]), which we do not analyse.

4.3.3. OpenAI Gym

The OpenAI Gym's Python library[16] [10], also referred to as `gym`, is by far the most popular Reinforcement Learning library, given its 34k+ stars on its GitHub repository[17] (as of September 2024). The library offers 30+ environments for experimenting with existing RL algorithms, testing new ones or simply playing. Among these environments, we can find some from Atari (e.g. Boxing, Pong) and control theory (e.g. Cart Pole).
A reason for its popularity is the possibility of seeing an agent operating graphically in any environment in the library. Figure 4.3 shows an example of such a view. Furthermore, the library is extremely simple to use: after the installation, a few lines of code are enough to run and view any environment. Every environment can be used through the same interface (i.e., methods), making switching between environments at any desired time effortless.

[13] `https://github.com/ray-project/ray/tree/master/rllib_contrib`: accessed September 2024
[14] `https://stable-baselines3.readthedocs.io/en/v2.3.2/guide/quickstart.html`: accessed September 2024
[15] `https://github.com/Stable-Baselines-Team/stable-baselines3-contrib`: accessed September 2024
[16] `https://www.gymlibrary.dev/content/basic_usage/`: accessed September 2024
[17] `https://github.com/openai/gym`: accessed September 2024

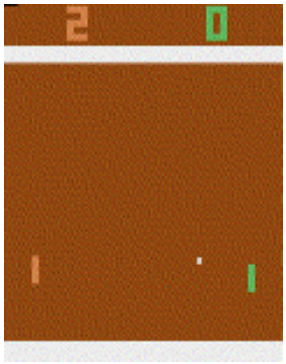

Figure 4.3.: Graphical view of the *Pong* environment. The image shows only one state, but when the environment is run, the user sees it evolving in a live window.

From version number `0.26.2`, the maintenance of the `gym` library has stopped, and another one has been created to replace it: `gymnasium`[18], `v0.29.0` at the time of writing. The new library APIs have not changed, making it easier for developers to switch. In this thesis, we are interested in investigating the usage of both libraries. The shared APIs make our work easier: every `gym` element we find interesting to analyse is present inside `gymnasium`, although the opposite may not be valid.

4.3.4. PettingZoo

PettingZoo[19] [63], `v1.24.2` as of September 2024, is the most widely used Python library to experiment with *Multi-Agent Reinforcement Learning* environments. It is a popular RL library, with 2.5k+ starts on its GitHub repository[20], but it is the standard for MARL.
The `gym` library deeply affects its design and usage. Like `gym`:

- It contains tens of environments to play with, offering a graphical view for each. Many of them, e.g. Atari, are extensions for multiple agents of `gym` environments.
- Functionalities are usable via specific elements (e.g. functions, classes, methods), which often have the same names as `gym`.

All this makes it easier for developers coming from `gym` to use `pettingzoo`, and also our analysis simpler.

[18] `https://github.com/Farama-Foundation/Gymnasium`: accessed September 2024
[19] `https://pettingzoo.farama.org/1.24.2/content/basic_usage/`: accessed September 2024
[20] `https://github.com/Farama-Foundation/PettingZoo`: accessed September 2024

4.3.5. GitLab workflows

GitLab workflows are an API allowing users to write automatic scripts inside any repository. They are created by defining a workflow structure in a specific YAML file: `.gitlab-ci.yml`. An example of a file is shown in Listing 4.1.

```yaml
image: gradle:8.1.1-jdk11-alpine

build-job:
  stage: build
  services:
    - docker:dind
  script:
    - gradle assemble
    - docker build -t $USER/$IMAGE_NAME .

test-job1:
  stage: test
  script:
    - gradle check --first-group

test-job2:
  stage: test
  script:
    - gradle check --second-group

deploy-prod:
  stage: deploy
  script:
    - docker login -u $USER -p $PW
    - docker push $USER/$IMAGE_NAME
```

Listing 4.1: Example of GitLab workflow to build, test and deploy a Java application.

A workflow can be imagined as a series of terminal commands run in a cloud machine offered by `gitlab.com` (or the specific GitLab installation when using GitLab on-premise). These commands are grouped into jobs, which are subsequently grouped into stages. Therefore, a GitLab workflow can be visualized as a graph. Note that, for better modularization and readability, GitLab workflows allow writing logic inside other YAML files besides `.gitlab-ci.yml`. To do so, one can import them with the `include`[21] keyword. In this thesis, we do not dynamically import such files. Furthermore, GitLab workflows may use shell scripts included in the repository to

[21] `https://docs.gitlab.com/ee/ci/yaml/index.html#include`: accessed September 2024

perform any type of command-line action. We do not take into consideration such scenarios. These choices were also made by Vassallo et al. [64] in their study on smells in GitLab workflow files.

4.3.6. GitHub workflows

GitHub workflows are the equivalent of GitLab ones but for `github.com`. They are defined by creating a YAML file under the `.github/workflow` folder.
Apart from using a different API, i.e., different keywords in the YAML file, there are other key differences between the counterparts. The ones relevant to our analysis are:

- More than one workflow can be defined by creating multiple files under the appropriate folder.
- Jobs are not grouped into stages; they all run in parallel by default.
- They can use so-called *GitHub actions*[22] to accomplish common tasks. These are publicly available and user-defined workflows that any other GitHub repository can employ. They are very commonly used and avoid the need to write a complex workflow by using logic defined by other users. Examples of functionalities offered by GitHub actions are setting up a Python environment[23], analysing PHP code[24], synchronising an AWS S3 bucket[25].

As for GitLab, GitHub workflows may use shell scripts in any of the jobs. Also, in this case, we do not consider the scenario. The same has been done by Calefato et al. [12] in their investigation on using MLOps on GitHub.

4.3.7. Docker

Docker[26] is by far the most used containerization technology among practitioners [3], eight times more popular than the second-place competitor Podman[27]. Given this huge difference, we focus entirely on Docker to investigate the images produced by a workflow.
To employ Docker inside workflows a CLI is used. After installation, it is enough to define a file with the image description and run specific commands from the terminal (e.g. `docker build`) to containerize an application. Section 5.1.9 details what we detect.

[22] `https://docs.github.com/en/actions/creating-actions/about-custom-actions`: accessed September 2024
[23] `https://github.com/actions/setup-python`: accessed September 2024
[24] `https://github.com/OskarStark/phpstan-ga`: accessed September 2024
[25] `https://github.com/jakejarvis/s3-sync-action`: accessed September 2024
[26] `https://www.docker.com/`: accessed September 2024
[27] `https://podman.io/`: accessed September 2024

5. Framework Architecture and Implementation

In this chapter, we explore the implementation of our proposed framework and, whenever possible, compare the choices made to those made in other studies. We then conclude with some details on how to execute our framework.

5.1. Detection Process

This section presents the elements we have chosen to identify the practices and sub-practices described in Chapter 4. Such elements can be considered sub-parts in source code (for Python) or workflow files (for GitLab/GitHub workflows). Examples of such elements are:

- a function call
- a class definition
- a job definition
- a job's step

The detection is binary, i.e. each element is either found or not. For each practice, we can have multiple elements signalling its presence. Finding one element is typically[1] enough for us to say that the practice is applied; therefore, the detection of a pattern or practice can be thought of as an OR of all its elements' detection results. Note that, for each searched element, it's possible to specify multiple characteristics that must be true (i.e. an AND). For example, we might look for function calls with a specific name and argument used.
To increase the understandability of our code, we have created *Detector* classes for each category of practices and patterns. More details can be found in Sections 5.4.2 and 5.4.3.

In this thesis, we have used two types of elements for the detection process: technology-based and heuristic-based.

- **Technology-based elements** are intended to find the usage of a practice or pattern using the specific API of a technology (i.e. a library, a tool). The

[1] There are examples of patterns and practices where more than one element must be present to detect the practice.

precision[2] of such elements is supposed to be high given that they are intended for a specific use case, but this also means that they generalise less, and have therefore a low recall[3]. Although they are limited to a specific technology, they can end up working for multiple ones if these share specific elements[4]. If this is the case, recall of technology-based elements is improved.

- **Heuristic-based elements** as the name suggests, use heuristics to detect a pattern or practice, looking for commonly used signatures (i.e. names). They are more general than technology-based elements because they are not limited to one use case. Nevertheless, being heuristics, it's hard to develop them, and the performance (i.e., precision and recall) may significantly change project by project.

We have relied on grey literature (i.e. tutorials, blogs, and technical documentation) to build the detectors over the analysed technologies. Such sources have been gathered using standard search engines such as Google, Bing, DuckDuckGo and search bars inside technical documentation. For each technology T and practice P, we used search keywords such as "*use P in T*", "*P tutorial for T*", and others. When searching inside technical documentation, we used the practice terms (e.g., "checkpoint" and "multi-agent"), and we navigated across the results. This process resulted in the definition of both technology-based and heuristic-based elements.
In the following subsections, we review all the practices' categories investigated in Chapter 4 to present *some*[5] of the elements used for detection. Table A.4 lists two example elements for each RL category.

Before delving into the following subsections, Table 5.1 summarises which technology, among those presented in Section 4.3, supports which RL practice category.

5.1.1. Checkpoints

As illustrated in Section 4.1.1, we are not interested in detecting checkpoint recoveries (i.e. loading an RL model) but only lines of code related to saving an agent. In our detector, we mostly rely on the following:

- Function/method calls and object creations.
- Function/method definitions.
- Assignments.

[2]The precision measures the proportion of how many elements detected as "positive" are truly positive.
[3]The recall measures the proportion of the positive elements found and the ones not found.
[4]In Section 4.3 we have seen how `gym` and `pettingzoo` name elements the same way. This is true also for other libraries.
[5]Given that for many patterns and practices, we use ten elements or more, not all are described. Nevertheless, it's enough to check out the `json` configuration files we use to know all of them

	Ray	SB3	Gym	PettingZoo
Checkpoints	✔	✔		
Hyperparameter Tuning	✔			
Distributed RL	✔	✔		
SARL	✔	✔		
MARL	✔			✔
RL Pipeline	✔	✔	✔	✔
Training Configuration	✔	✔	✔	✔

Table 5.1.: Summary of which Python library can apply which RL practice category. Note that in this table we consider all the `ray` modules and not only `RLlib`.

Since checkpoints are related to an agent, most technology-based elements are found for `stable-baselines3` and `RLlib`. Searches for `gym`, `gymnasium`, and `pettingzoo` yield only one result related to `agilerl`, an RL library not analysed in this thesis. Therefore, we did not include that result in our analysis.
To create heuristic-based elements, we have looked for signatures such as *create_checkpoint*, *perform_checkpoint*. We did not use simpler signatures, such as *checkpoint*, as these may detect loading from a file instead of saving to a file.

In addition to detecting the practice, Section 4.1.1 described two other patterns: checkpoint path and checkpoint frequency. We use function/method calls, keywords, and string constants to detect them.

5.1.2. Hyperparameter Tuning

For the Hyperparameter tuning best practice, we limit ourselves to the elements that run a tuning process: function/method calls and function/method definitions. All configurations related to tuning are ignored.

We have found only three technology-based elements to detect, all inside `ray.tune`. In addition, we have defined heuristics which look for elements like *do_tuning*, *tuning_run*.

5.1.3. Distributed Reinforcement Learning

Besides detecting the creation of distributed agents, we are interested in other things relevant to the pattern, such as the configuration of workers.

As for the checkpoints, distributed reinforcement learning is deeply tied to the algorithm library. Therefore, the elements we are looking for come from RLlib and stable-baselines3. As explained in Section 4.1.3, whether an RL algorithm applies distribution strategies heavily depends on the specific implementation. For this reason, we have relied on the documentation to determine the distributed nature of an implementation. Table A.2 summarises our findings.
For stable-baselines3, all algorithms support multi-processing[6], using a library-specific solution referred to as *vectorized environments*. Even though thirteen algorithms apply distributed RL, we are interested in only seven because the others are part of the package `sb3-contrib` and not the official `stable-baselines3`.
Conversely, RLlib doesn't implement all its algorithms in a distributed fashion: out of the nine implemented algorithms, only six do.

In the detector, we look for the following elements:

- Function/method calls and object creations.
- Function/method definitions.
- Assignments.
- String constants.

The need for string constants arises from an API used in `ray.tune`, where an algorithm is instantiated based on its string name.

For this pattern, we used fewer heuristic-based elements because, as previously illustrated, the distributed nature of an algorithm mostly depends on the implementation. Looking for specific algorithm names (e.g. *DQN*) may result in a true or false positive based on the particular use case. For this reason, we primarily relied on technology-based elements.

5.1.4. Single-Agent Reinforcement Learning

The main elements we have crafted to detect usages of Single-Agent RL are:

- Gym, Gymnasium, PettingZoo and RLlib[7] environments.
- RLlib and Stable-baselines3 agents.

Given that an agent or environment can be related to a Multi-Agent setting, we have distinguished between *certain* SARL elements (e.g. from gym) and *potential* SARL elements. To determine whether the latter are instances of the Single-Agent pattern, rather than MARL, the `SARLDetector` class also looks for Multi-Agent technology-based and heuristic-based elements.

[6] `https://stable-baselines3.readthedocs.io/en/v2.3.2/guide/algos.html`: accessed September 2024
[7] RLlib offers many wrappers, rather than executable environments.

5.1.5. Multi-Agent Reinforcement Learning

The detection of the MARL category is based on identifying specific environments and algorithm configurations because both must be pre-disposed to work with more than one agent. Among the 4 libraries we analysed, we found references to multiple agents only for `RLlib` and `PettingZoo`, while nothing for `stable-baselines3` or `gym`. This is because both `gym` and `stable-baselines3`[8] were specifically created for single-agent RL.

We can now list the type of elements we mostly rely on for the detection:

- Creation of a PettingZoo predefined environment or definition of a new one extending PettingZoo environment classes.
- Configuration of an RLlib algorithm to use MARL.
- Creation and usage of RLlib classes specifically designed for MARL.

Similarly to SARL, we have identified elements signalling *certain* use of the Multi-Agent pattern and others signalling *potential* use.

Besides identifying the usage of MARL, we also want to determine characteristics like the type of environment and the number of agents. To do so, we first went through each PettingZoo environment and categorised it. The result of this investigation is shown in the appendix in table A.3.
Identifying these characteristics entirely relies on the ID of the PettingZoo environment detected.

5.1.6. Reinforcement Learning Pipeline

As described in Section 4.1.5, we are interested in identifying these steps of an RL program:

- Definition of a new environment.
- Creation of an environment.
- Creation of an RL agent.
- Training of an RL agent.
- Saving an RL agent.
- Evaluating an RL agent.

We can rely on the elements identified in other categories for the environment definition, environment initialisation, algorithm initialisation, and model-saving steps. Therefore, new elements have been added only for the training and evaluation step.

To recognise these two phases, the main elements we identify are:

[8] `https://araffin.github.io/post/sb3/`: accessed September 2024

- **Model Training**:
  - Method calls.
  - Function definitions.
  - Workflow jobs[9].
- **Model Evaluation**:
  - Method calls.
  - Function definitions.

It must be noted that, like for the Hyperparameter Tuning best practice, we have chosen to limit the detection process to the elements *actually* running the training, saving and evaluation steps. We have ignored all elements simply configuring these steps.

5.1.7. Training Configuration

In Section 4.1.6, we have described the two key configurations we are interested in identifying. These are:

- Hyperparameters usage.
- Seeding.

We now list the main elements we use to identify these two configurations:

- **Hyperparameters usage**. In this case, We look for the names of each hyperparameter inside:
  - Assignments.
  - Keywords.
  - String values.
  - Functions/methods definitions.
  - Functions/methods input parameters.
- **Seeding**:
  - Function and method calls of major libraries[10] which offer setting seeds.
  - Keywords.
  - Functions/methods definitions.
  - Functions/methods input parameters.

[9] As stated in Section 4.1.5, we consider Continuous Training as part of the RL training step. This is identifiable from workflow jobs.

[10] Beside gym and pettingzoo having seed functions and methods, we also consider other libraries' functions (e.g., `numpy.random.seed`)

5.1.8. Workflow Automation

The detection of the automation of a process in a `GitLab/GitHub` project is trivial as it is enough to find any job defined in a workflow file. A job is defined as:

- Any top-level clause which uses the `script` keyword[11] for GitLab.
- Any clause under the `jobs` keyword[12] for GitHub.

After identifying jobs, we are interested in grouping them. The grouping is done for GitHub jobs based on the workflow (i.e., the file under `.github/workflows`) since multiple workflows can be defined per project. Conversely, in GitLab only one workflow file is allowed but jobs can be grouped by stages[13][14]. Therefore from GitLab workflows we also extract the `stages` top-level keyword.

Besides extracting and grouping jobs, we are also interested in identifying the jobs' dependencies. Both GitLab[15] and GitHub[16] use the `need` clause to specify jobs' dependencies. Nevertheless, these are explicit dependencies and do not consider the *dependencies' dependencies*[17]. In this thesis, we walk through the jobs graph to identify all cumulated dependencies, direct and indirect.

5.1.9. Workflow Results

As previously said in this chapter, we are interested in two types of workflow results: job artifacts and images pushed.

The recognition of job artifacts is straightforward for both GitLab and GitHub. Using the relative documentation, we can see that GitLab artifacts can be extracted from the `artifacts:paths`[18] clause. Instead, on GitHub, artifacts are defined inside a job's step, using the action `upload-artifacts`[19], with the specific paths defined under `with:path`.

We have two methods to detect images pushed: one for GitLab/GitHub and one only for GitHub. The docker CLI works for both workflows; whenever we find the `docker push` or `docker image push` commands, the workflow produces a Docker

[11] `https://docs.gitlab.com/ee/ci/jobs/`: accessed September 2024
[12] `https://docs.github.com/en/actions/writing-workflows/workflow-syntax-for-github-actions#jobs`: accessed September 2024
[13] `https://docs.gitlab.com/ee/ci/yaml/index.html#stages`: accessed September 2024
[14] `https://docs.gitlab.com/ee/ci/yaml/index.html#stage`: accessed September 2024
[15] `https://docs.gitlab.com/ee/ci/yaml/index.html#needs`: accessed September 2024
[16] `https://docs.github.com/en/actions/writing-workflows/choosing-what-your-workflow-does/using-jobs-in-a-workflow#defining-prerequisite-jobs`: accessed September 2024
[17] If Job C defines Job B as a dependency and Job B defines Job A, then Job A is an implicit dependency of Job C
[18] `https://docs.gitlab.com/ee/ci/yaml/index.html#artifacts`: accessed September 2024
[19] `https://github.com/marketplace/actions/upload-a-build-artifact`: accessed September 2024

image and pushes it to a container registry. The second element we can detect is the action `docker/build-push-action`, which is available only for GitHub workflows.

5.2. Insights

Before describing our proposed framework, we illustrate the insights produced when a repository is analysed. Figure 5.1 displays the class diagram of the insights. Overall, we aim to provide two types of results:

- **Where a practice or pattern is applied**, showing the location (i.e., the source lines) in the source code.
- **How a practice category is applied**, indicating the specific patterns and best practices identified.

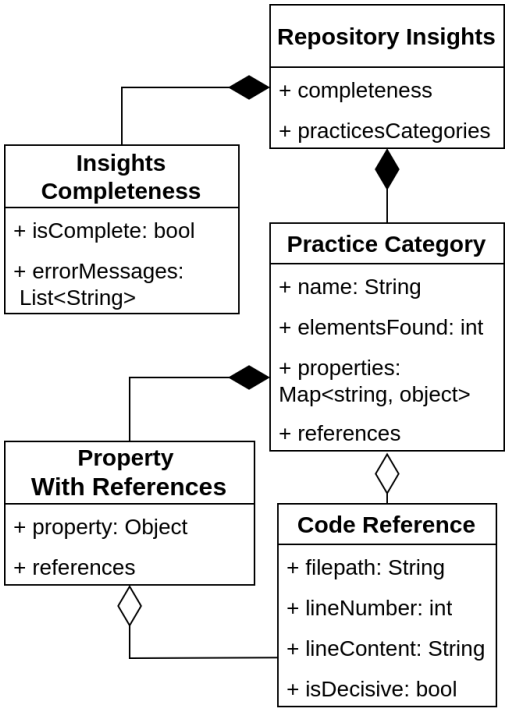

Figure 5.1.: Class diagram for the insights produced by our framework.

For this reason, we have created the `RepositoryInsights` class which contains:

- `PracticeCategory` objects to inform the user where and how patterns and practices are applied.
- `InsightsCompleteness` object, which has information about the occurred errors. Since parsing and analysing source files can often lead to bugs[20], we

[20] Source files can often be syntactically incorrect, which may raise errors during parsing. Similarly,

catch any exception thrown for a specific file and show corresponding error messages to the user. The exceptions are not propagated to the analysis of other files.

To show *where* a pattern or practice is applied, we use the `CodeReference` class. This class contains information on the source code (i.e. file path, line number and line content) and whether the specific line of code was *decisive* for the detection. The `isDecisive` fields allows adding additional lines of code to give the user more context. For example, when a class is identified as related to a pattern or practice, all the class' lines of code can be returned, leaving only the top ones as *decisive*.
Starting from a practice category, we use the `properties` field to list which exact patterns and practices are applied and specify details over them[21].
Given the many different types of details that may be shown to the user, we have chosen to use a generic `map: string -> object`, which may have anything as a value. Although this raises complications when the insights are to be shown, it allows us to use a common class across our framework.
Besides knowing where a pattern or practice is used, a practitioner may be interested in knowing where a detail (e.g. a job) is detected. For this reason, we have created the class `PropertyWithReferences`, which couples a property (of any type) to where it was found (i.e. using `CodeReference` objects).

In addition to all these insights, we save the number of elements found for a specific practice category. We have chosen not to use one per specific pattern and practice, to keep the insights data structure simple. It's important to note that this number does not coincide with the number of `CodeReference` objects for two reasons:

- There can be multiple elements found on one line. For example, the line
  `checkpoint_path = CheckpointConfig().path`
  has two elements relevant to the checkpoint practice.
- There can be elements spanning multiple lines of code (e.g. an assignment).

5.3. Framework Architecture

This section briefly summarises the framework's architecture, describing each application and how they interact. In the next section, we provide in-depth depictions of how the components work and the key design decisions for each of them.

Figure 5.2 shows a high-level view of the architecture of the framework proposed in this thesis. We can see four elements:

the files can be syntactically incorrect, but the parser does not expect an *edge case*.

[21] As explained in Section 5.1, for MLOps patterns and practices, we also want to extract and display keywords found in the source code and not only show specific lines of code.

- Angular application, mostly containing user forms and the *Insights Displayer*.
- Java application, mostly containing the *Repositories Manager*, *Repository Analyser*, *Python Analyser Facade* and *Workflows Analyser*.
- Python application, mostly containing the *Python Analyser*.
- Repositories volume.

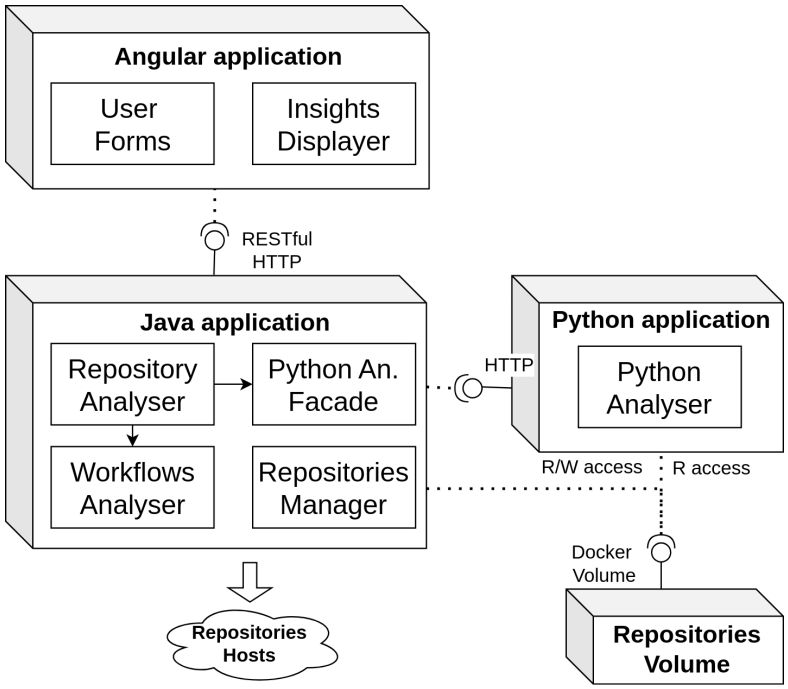

Figure 5.2.: Architecture of the framework. Note that only some key components are displayed.

Considering our goals, finding a single programming language for the analysis is impossible as no programming language offers all the needed libraries. Therefore, multiple analysers are necessary.
We have chosen to use a frontend application to display the insights to the user. In general, analysis tools can be interfaced in multiple ways, among which we can find a command line interface (CLI) [14, 64, 58], a graphical user interface (GUI), an IDE extension [22], a library, an HTTP server [47], etc. Given that our target is ultimately a human operator and we must display a large quantity of information for several practices, we chose to display results using a running front-end application to access via a web browser at `localhost:4242`.
We containerised each application to simplify deployment and future usage of our framework. The framework is, therefore, locally deployable with a `docker-compose.yml` file. To allow the Analysers to access the repository files, we use a docker volume[22].

[22] `https://docs.docker.com/storage/volumes/`: accessed September 2024

The *Repositories Manager* writes in it, while the Analysers only read. More details about the replication package can be found in Section 6.5.

To understand the framework's functioning, we can describe an example of an analysis process. Figure 5.3 shows the corresponding activity diagram. The process starts with the user inserting the analysis options (i.e. the repository to inspect) in the Angular Application. If this is not already on disk, an HTTP request is sent to the *Repositories Manager* to fetch this repository. After a successful fetch, the UI sends an analysis request to the Analysers Manager. At this point, the *Repository Analyser* calls the two Analysers (i.e. the *Python Analyser Facade* and the *Workflows Analyser*) and combines the obtained results. Finally, the Angular Application receives and processes the results for optimal visualization. Now, the user can view all the information and filter it. The user can then start again the analysis process for another repository if desired.

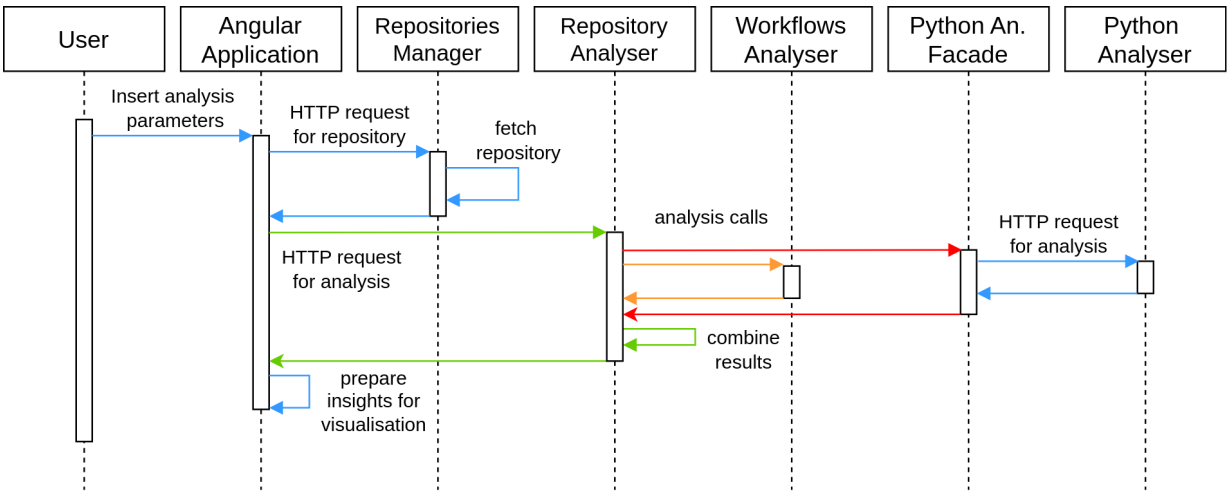

Figure 5.3.: Simplified activity diagram for the analysis process. In this scenario, we suppose the repository is to be fetched and that both Analysers do not fail. Note that superfluous intermediate logic (e.g. *RequestHandler* classes) is removed.

5.4. Framework Components

This section describes how the main parts of our framework work and shows example code snippets taken from them.

5.4.1. Repository Analyser

The Repository Analyser is a component whose job is to handle analysis requests by forwarding them to the *Workflows Analyser* and *Python Analyser Facade*. After these have been returned, the results are merged. The logic to do it is trivial; Listing 5.1 shows the Java code.

```
public static RepositoryInsights merge(RepositoryInsights... insightsArray){
    RepositoryInsights mergedInsights = new RepositoryInsights();
    InsightCompleteness mergedCompleteness = new InsightCompleteness();
    Arrays.stream(insightsArray)
        .filter(insights -> !insights.isEmpty())
        .forEach(insights -> {
          mergedInsights.practicesCategories.addAll(insights.practicesCategories);
          mergedCompleteness.addIncompletenessMessages(
            insights.completeness.getIncompletenessMessages()
          );
          mergedCompleteness.setComplete(
            mergedCompleteness.isComplete() && insights.completeness.isComplete()
          );
    });
    mergedInsights.setCompleteness(mergedCompleteness);
    return mergedInsights;
}
```

Listing 5.1: Java code to merge `RepositoryInsights` objects.

5.4.2. Python Analyser

The Python Analyser, as previously outlined, is a component written in Python used to analyse Python files employing the `ast` library[23]. In Section 5.3, we have already seen how the Python Analyser is integrated within our framework and how to contact it. Therefore, in this section, we describe how we find the elements presented in Section 5.1. Figure 5.4 shows a simplified class diagram containing the main classes for the analysis process.

Before describing the key components inside the Python Analyser, it is necessary to explain how the `ast` package works. `ast` is a library part of the official Python ecosystem, and it comes inside any Python interpreter (i.e., no need for installation). It is used by interpreters to parse Python source code, and therefore, it is extremely reliable. As new functionalities are added to the Python programming language, the library evolves to interpret all of them. For these reasons, it is commonly used in research [14, 58, 47, 56] and is employed in this thesis.
Another reason for us to use `ast` is that it's possible to know the position (i.e. starting/ending line/column) of every element in a source file, which is required for our

[23] `https://docs.python.org/3/library/ast.html`: accessed September 2024

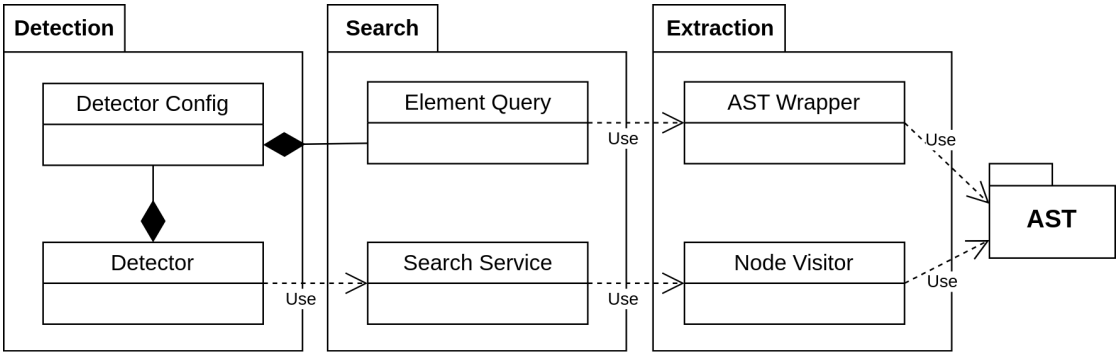

Figure 5.4.: Simplified class diagram of the Python Analyser. Two packages have been removed from the view: `insights` contains the classes introduced in Section 5.2, `analysis` manages all the `Detector` classes.

study.

The library parses *logical* components from a given Python source code into specific classes. Examples of these components include a function definition, a method call, an assignment, etc. Each of them is associated with one or more classes inside AST. Given the rich syntax of Python, there are tens of classes, but for this thesis, we are interested in only a few of them. The complete list is in table 5.2.
Once the library parses the source file, it is possible to find every object of the desired type. From here, obtaining most of the desirable information is possible by accessing specific object fields. An example of usage of `ast` is shown in Listing 5.2.

```
import ast

functions = []
class FunctionDefVisitor(ast.NodeVisitor):
    def visit_FunctionDef(self, node):
        functions.append(node)
        self.generic_visit(node)

with open("file.py", 'r') as file:
    source_code = file.read()

tree = ast.parse(source_code)
visitor = FunctionDefVisitor()
visitor.visit(tree)
for function_def in functions:
    print(function_def.name)
```

Listing 5.2: Python script to extract function definitions from a source file and print all the function names.

Even though the benefits of the library are many, it also has a few downsides relevant to our study:

- The parsing phase fails if an analysed source file applies Python keywords or constructs not recognized by `ast`. There are two scenarios where this can happen:
  - The source file is syntactically incorrect. An error found on any line would prevent any other line from being analysed.
  - The source file uses any functionality introduced from Python `3.11`. Since the Python Application uses Python `3.10`, new functionalities would be considered syntax errors by `ast`.
- The library is not easy to use and does not offer a high-level API to interact with source code elements. Extracting information requires a complex interaction with the library class fields. This is why we have created a `ASTWrapper` class that is presented in the next paragraphs.

Considering both these downsides, there's a chance for each file to be parsed or analysed erroneously. Any error may halt the analysis phase and prevent insights from being returned to the user. For this reason, whenever a `Detector` fails on any file, the error is caught, and no analysis is performed by that `Detector` on that file. Note that this does not stop other `Detectors` from analysing the said file. Error messages are shown via the UI to inform users of such occurrences.

Another important design decision for the Python Analyser is on source file dependencies. Our static analysis does not load all the source file's dependencies (i.e., imports). Instead, we extract all the possible information from the *explicit* imports. This choice was made to simplify our analysis significantly. In practical terms, our analysis does not take into consideration:

- Wildcard imports, e.g. `from library import *`.
- Source code imported from others of the user's source files (except the imports themselves).

This design decision is also present in the work of Pradel et al. [56] for their type inferring tool. The consequences of this choice are the following:

- It is impossible to know the full identifier of the elements imported with wildcards. This means, among other things, that we might not recognize a library class instantiation relevant to a pattern or practice.
- It is impossible to know all the user's logic of a source file because part of this logic is imported by another source file. An example of the effects of this scenario is shown in listings 5.3 and 5.4.

```
import library


def step1():
  library.init()

def step2():
  library.exec()

step1()
step2()
```

Listing 5.3: Example of recognised two-step pattern. In this case, all steps are done in the same source file.

```
import library
from . import exec_file

def step1():
  library.init()

def step2():
  exec_file.exec_function()

step1()
step2()
```

Listing 5.4: Example of not recognised two-step pattern. In this case, not all steps are done in the same source file.

In the following paragraphs, we describe the key classes of the Analyser, displayed in Figure 5.4.

NodeVisitor

To extract the desired elements from a Python source file, it is necessary to extend the `ast.NodeVisitor` class and override specific methods. An example of this is shown in Listing 5.2. For each element we are interested in (e.g. `ast.FunctionDef`), we have created a specific `Visitor` class.

ASTWrapper

We have created a wrapper class for each `ast` class we are interested in analysing. It was done because, as previously mentioned, obtaining relevant information from source code elements is not so trivial. Listing 5.5 is an example of this. Encapsulating all this logic behind a method improves the understandability of the Python Analyser.

```
def get_import_identifiers(import_from: ast.ImportFrom):
  identifiers = []
  for name in import_from.names:
    if name.name != "*":
      if import_from.module is not None:
        identifiers.append(import_from.module + "." + name.name)
      else:
        identifiers.append(name.name)
    else:
      identifiers.append(None)
```

```
11  return identifiers
```

Listing 5.5: Python function to get the full qualifiers from an import statement. For this use case, it is required to access multiple `ast` fields and write a non-trivial case management logic.

Table 5.2 shows all the wrapped classes used in the Python Analyser and the information we extract from such classes. These were defined based on the elements we want to detect, listed in Section 5.1.

Note that even if much information can be extracted using the `ast` library, this does not allow us to answer some relevant questions. For example:

- Whether a call is a method call, function call or object instantiation. Given a call, we can distinguish between `call()` and `something.call()`, but we cannot know whether `something` is an object or a module. For our analysis, we assume that every call using the dot operator is a method call.
- The type of a variable. This is caused by Python using duck-typing, leading to variable type being inferred at runtime. Research for Python type inference tools, capable of receiving a Python source code and statically inferring the types of variables, is in progress [47, 56]. Still, at the current time, the tools can only infer basic types (e.g. `int`, `str`). Consequently, in this thesis, we do not try to filter method calls based on the object type but only on the method characteristics (i.e. name, arguments, etc.).

ElementQuery

In Section 5.1, we have listed all the source code elements we are interested in detecting. We have created a base `ElementQuery` class and extended it for each of the `ASTWrapper` classes to describe them systematically. The `ElementQuery` class filters the elements among the ones returned by `NodeVisitor` classes.
`ElementQuery` offers the `accept_element` method that receives an `ASTWrapper` instance and compares the `ASTWrapper` fields with the characteristics (e.g. method name). This comparison is done only after ensuring the `ASTWrapper` object is of the expected type: e.g. `KeywordQuery` expects to receive `KeywordWrapper`.
A description of all the specifiable characteristics is shown in Table 5.3. These are very similar to the information extracted from each `ASTWrapper`, presented in Table 5.2.

Note that we have not created an `ElementQuery` for each element presented in Section 5.1, as we have often used regular expressions to describe multiple technology-based and heuristic-based elements at once. For example, the detection of `stable_baselines3.Algorithm` and `stable_baselines3.Algorithm.load` have been merged into one `ElementQuery`: `stable_baselines3.Algorithm(\.load)?`.

Wrapper class	Description	Wrapped around
`Assignment Wrapper`	Wraps around all assignments. From them, we extract: • targets name (e.g. `x = y = func()`), regardless of the complexity of the target: arrays, dictionaries, etc.	`ast.Assign`, `ast.AnnAssign`
`Call Wrapper`	Wraps around method calls, function calls and object instantiation. From them, we extract: • function/method/class name • whether it is a method call. It uses a heuristic: if the dot operator is used, it is considered a method call. • (when possible) full call name (e.g. `obj.method`). • (when applicable) identifier if it is a function/class imported. • arguments values. • keyargs used. • argument positions used.	`ast.Call`
`Class Def Wrapper`	Wraps around class definitions. From them, we extract: • class name. • (when applicable) superclasses names. • (when applicable) full identifiers of the superclasses, if these are imported.	`ast.ClassDef`
`Function Def Wrapper`	Wraps around function and method definitions. From them, we extract: • function/method name. • arguments names.	`ast.FunctionDef`, `ast.AsyncFunctionDef`
`Imported Element Wrapper`	Wraps around `import` and `from ... import`. Note that every import statement may have multiple imported elements and that we ignore wildcard imports. From non-wildcard imports, we extract: • import identifier. • import usage keyword, i.e. how the import can be used in the file.	`ast.Import`, `ast.ImportFrom`
`Keyword Wrapper`	Wraps around keywords for calls. From them, we extract: • key name. • key value.	`ast.keyword`
`String Constant Wrapper`	Wraps around all strings in a file, not only the ones assigned to a variable. From them, we extract: • string value.	`ast.Constant`

Table 5.2.: Wrapper classes defined in the Python Analyser, alongside the characteristics they extract from `ast` fields.

Query class	Available Querying Options
`Assignment Query`	• target: regex string. If the assignment is multi-target (e.g. `x = y = 2`), it applies an OR on each target.
`Call Query`	• name: regex string. • name_case_sensitive: boolean. False by default. • is_method: boolean. • identifier: regex string. • identifier_case_sensitive: boolean. True by default. • required_arguments: list of tuples *[argument position, argument name]*. It contains all the arguments a call must have. • missing_arguments: list of arguments a call must not have.
`Class Def Query`	• name: regex string • name_case_sensitive: boolean. True by default. • superclass_name: regex string. If there is multiple inheritance, it applies an OR on each superclass name. • superclass_name_case_sensitive: boolean. True by default. • superclass_identifier: regex string. If there is multiple inheritance, it applies an OR on each superclass identifier. • superclass_identifier_case_sensitive: boolean. True by default.
`Function Def Query`	• name: regex string. • required_parameters: list of string regexes. It contains all the parameters a function must have.
`Keyword Query`	• name: regex string.
`String Constant Query`	• value: regex string.

Table 5.3.: Query classes defined in the Python Analyser, alongside the filtering options they offer.

DetectorConfig

In Section 5.1, we have presented all the elements we look for inside Python files. To make our framework easily extensible, we have used `json` configuration files, which describe all the elements a Detector looks for. The configuration files are then, at runtime, deserialized into `DetectorConfig` objects. Internally, the `DetectorConfig` contains `ElementQuery` instances.
Note that in the `json` files, we have included all the links to each technology-specific element. The links are added using json5 style[24] and are automatically removed before parsing.

A `DetectorConfig` object is composed of multiple `Collections` to more easily group query objects. To distinguish the type of `ElementQuery`, each collection is further divided into *query types*. Figure 5.5 depicts the structure of a `DetectorConfig` object while Listing 5.6 shows an example of `DetectorConfig` in `json` form.

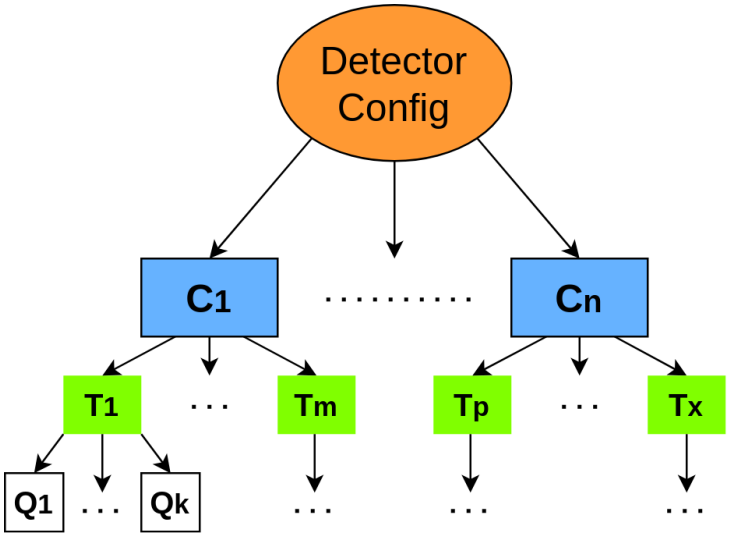

Figure 5.5.: A `DetectorConfig` object visualized. We can find `n` collections, internally divided into query types, each containing a certain number of `ElementQuery` objects.

```
1 {
2   "collection-ray-tune-1": {
3     "calls": [
4       { "identifier": "ray\\.tune\\.run$"},
5       { "identifier": "ray\\.tune\\.run_experiments$"}
```

[24] `https://json5.org/`: accessed September 2024

```
    ]
  },

  "collection-ray-tune-2": {
    "calls": [
      { "identifier": "ray\\.tune\\.Tuner$"},
      { "name": "fit$", "is_method": true}
    ]
  },

  "collection-heuristics": {
    "functions definitions": [
      { "name": ".*(do|mak|perform|run).*(tune|tuning).*"}
    ],
    "calls": [
      { "name": ".*(do|mak|perform|run).*(tune|tuning).*"}
    ]
  }
}
```

Listing 5.6: Example of `json` file containing a `DetectorConfig` object for the Hyperparameter tuning practice. We can see three collections, and for each collection, one or more query types with inside `ElementQuery` objects. Note that the comments added to each technology-based query object, linking to the corresponding documentation, have been removed from the listing for better visualisation.

We have created collections primarily to group query objects based on the specific pattern or practice they intend to detect.

Search Service

The `SearchService` obtains all the desired elements from a source file. Given a `DetectorConfig` object and source file, it returns a `SearchResults` object, which follows the same structure of the received `DetectorConfig`: collection → query type → `ResultSummary`. Instead of a `ElementQuery` object like for `DetectorConfig`, a `ResultSummary` is returned. This contains:

- The elements found for the corresponding `ElementQuery`.
- The number of unique found elements.
- The code references for the found elements.

We have chosen this structure so that any `Detector` can select which results it is interested in filtering by collection(s), query type(s) and/or `ElementQuery` index.

Primarily, the filtering is done based on the collection, as one collection is typically associated with a specific pattern or practice.

Detector

The logic for the detectors is pretty small because most of it is contained inside the other classes of the Python Analyser. Furthermore, the logic is simplified by a key design decision: a project's analysis is done by merging all the single files' analyses.

Given this design decision, a `Detector` only needs to offer one method: `analyse_file`. Listing 5.7 provides an example of the implementation of this method.

```
def analyse_file(self, tree) -> PracticeCategory:
        results = perform_search(tree, RLPipelineDetector.configs)
        result_summary = results.get_summary()

        if result_summary.found_elements_number == 0:
            return PracticeUsage(Practice.RL_PIPELINE)
        else:
            return PracticeUsage(Practice.RL_PIPELINE,
                    found=result_summary.found_elements_number,
                    found_references=result_summary.code_references,
                    properties=RLPipelineDetector._get_properties(results)
            )
```

Listing 5.7: Implementation of the `analyse_file` method for the *RL Pipeline Detector*. Part of the logic is implemented in the method `_get_properties`, not shown here.

As we can see from Listing 5.7, the `analyse_file` method returns a `PracticeCategory` object. All these objects[25] are collected by utility classes and added to a list.

5.4.3. Workflows Analyser

The Workflows Analyser is a Java component and part of the Java Application. It is used to analyse YAML workflow files, relying on two parsing libraries: Jackson[26] and SnakeYAML[27]. In Section 5.3 we have already seen how the analyser is integrated into the framework. Therefore, in this subsection, we focus on the inner workings of the Workflows Analyser.
We have proceeded with an approach similar to the Python Analyser to develop this component. However, the logic is considerably more straightforward, given fewer

[25] the number of `PracticeCategory` objects instantiated is $N_{detectors} * N_{pythonfiles}$
[26] `https://github.com/FasterXML/jackson`: accessed September 2024
[27] `https://bitbucket.org/snakeyaml/snakeyaml/src`: accessed September 2024

practice categories and elements to detect inside workflow files. Figure 5.6 shows a simplified class diagram containing the main classes for the analysis process. Each of these classes, and related ones not displayed in the class diagram, is described in the following paragraphs.

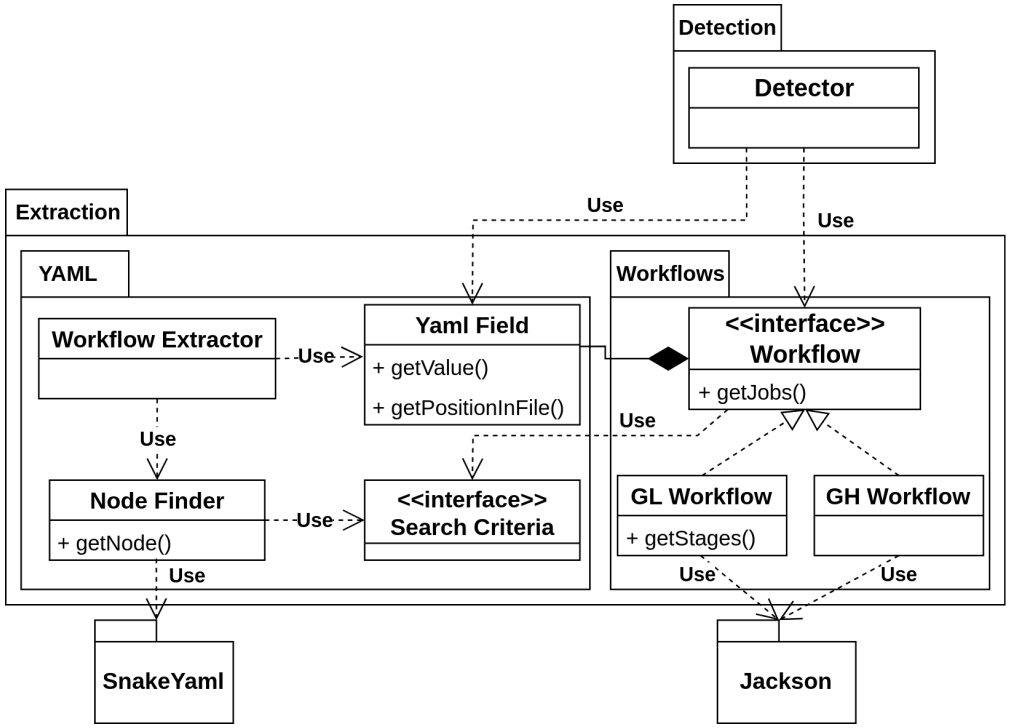

Figure 5.6.: Simplified class diagram of the Workflows Analyser. Some classes have been removed for higher understandability but are mentioned in the Workflows Analyser subsection.

YamlField

As previously described in Section 5.2, we need to display to the user where specific information was found in the analysed files. In the Python Analyser, the `ast` library already offered this functionality. This is not the case for the parsers used in the Workflows Analyser, therefore we have created the YamlField class which holds any kind of data (using Java generics[28]) and the position (starting/ending line) of the data in the file.

Workflow

Similarly to the `ASTWrapper` abstract class for the Python Analyser, we have created

[28] `https://docs.oracle.com/javase/tutorial/java/generics/types.html`: accessed September 2024

a `Workflow` interface, which is then implemented by two classes: `GLWorkflow` and `GHWorkflow`. These classes contain fields and methods to access information over the parsed workflows. Other classes and interfaces (not shown in Figure 5.6) represent workflow data, such as `Job`, `GLJob`, `GHJob`, `GHStep`, etc. Technology-specific (GitHub and GitLab) classes are necessary for two reasons: the different YAML structures in the files, and the different data contained in the files (e.g., only GitLab workflows specify stages). Nevertheless, to simplify the logic for the `Detector` classes, we use interfaces (e.g. the `Job` interface) whenever possible.

We have used Jackson to automatically serialise YAML keywords into class fields for the classes inside the `workflow` package. Jackson finds and extracts the desired keywords by simply annotating *setter* methods. An example of this usage is shown in Listing 5.8. It's important to note that, for some YAML keywords, it's necessary to do the casting logic by hand whenever a keyword can be of multiple types. An example is the `needs`[29] keyword for GitLab jobs that can be a `string`, a `list` of `strings` or a `list` of `maps`.

The Workflow classes, and the related ones, also hold the location of the YAML values inside `YamlField` instances. They use the `NodeFinder` and `SearchCriteria` classes to find these locations, which internally use the SnakeYAML library. This is necessary because Jackson does not offer this functionality.

```
1  public class GLJob implements Job {
2    //...fields
3
4    @JsonSetter("stage")
5    public void setStage(String stage) { ... }
6
7    @JsonSetter("artifacts")
8    public void setArtifact(GLArtifact artifact) { ... }
9
10   @JsonSetter("before_script")
11   public void setBeforeScript(List<String> beforeScript) { ... }
12
13   @JsonSetter("script")
14   public void setScript(List<String> script) { ... }
15
16   @JsonSetter("after_script")
17   public void setAfterScript(List<String> afterScript) { ... }
18
19   @JsonSetter("needs")
20   public void setNeeds(Object needsObj) { ... }
21
22   //...getters
23
```

[29] `https://docs.gitlab.com/ee/ci/yaml/index.html#needs`: accessed September 2024

```
24   //...other methods
25
26 }
```

Listing 5.8: Example of usage of the Jackson library inside the `GLJob` class. Fields, other class methods and the setting logic have been removed for higher understandability.

NodeFinder

`NodeFinder` is a utility class that finds the nodes of YAML keywords in a file. Given a tree (i.e. the result of SnakeYAML parsing) and a list of `SearchCriteria` instances, it *walks* the tree using the criteria until it finds the final node. Then, using fields inside SnakeYAML classes, it's possible to get the location data of the node.

SearchCriteria

`SearchCriteria` is an abstract class used as a common superclass for three types of criteria: `SearchInMap`, `SearchElementInArray`, `SearchRangeInArray`. Using a specific sequence, it's possible to find any desired YAML element(s).
As an example, to find the first step of the *test-job2* of Listing 4.1, we can define the sequence `SearchInMap("test-job2")`, `SearchInMap("script")`, `SearchElementInArray(0)`.

WorkflowExtractor

We have employed the `WorkflowExtractor` class to manage the parsing process. This is simply a *facade*, and it relays the logic to technology-specific classes (`GLWorkflowExtractor` and `GHWorkflowExtractor`) based on the workflow to parse. Inside these classes, `Workflow` objects are instantiated. Jackson sets the fields automatically, while the field locations are set using the `NodeFinder` class.

Given the complexity of parsing, any YAML field may fail to be correctly processed (e.g., if it's an unexpected type). Any error may halt the whole analysis process, making the Workflows Analyser not resilient. For this reason, any exception when parsing a workflow is caught and does not propagate to other workflows. Furthermore, the same is true for jobs inside a specific workflow.

Detector

As for the Python Analyser, the Workflows Analyser has a `Detector` interface implemented into a class for each practice category to analyse. Differently from the Python Analyser, the detection logic is simpler because far fewer workflow elements have to

be analysed. The Java Application does not replicate all the helper classes inside the Python Analyser, such as `DetectorConfig` and `SearchService`. All the elements to detect are specified inside the source code instead of `json` configuration files.

The `Detector` interface offers only the `analyseWorkflow` method, which inputs a Workflow and some analysis parameters (e.g., whether the analysis is for GitHub or GitLab). After the analysis, it returns a `PracticeCategory` object. All these objects[30] are collected by utility classes and added to a list. Listing 5.9 shows an example implementation of the `analyseWorkflow` method.

```
public PracticeCategory analyseWorkflow(Workflow workflow, AnalysisParameters
    analysisParameters){
  Map<String, Object> properties = new HashMap<>();
  properties.put("results", new ArrayList<>());

  PracticeCategory practiceCategory =
  new PracticeCategory(Practice.ARTIFACTS_PRODUCED, 0, new ArrayList<>(), properties);

  analyseJobArtifacts(workflow, practiceCategory);
  analyseDockerArtifacts(workflow, practiceCategory);

  if(properties.get("results").equals(new ArrayList<>()))
    properties.remove("results");

  return practiceCategory;
}
```

Listing 5.9: Implementation of the `analyseWorkflow` method for the *Workflow Results* detector. Part of the detector's logic is implemented in the methods `analyseJobArtifacts` and `analyseDockerArtifacts`, not shown here.

5.4.4. Angular Application

The Angular application is a frontend application accessible at `localhost:4242`. It has two goals: receiving as input the analysis parameters (i.e. the repository to analyse) and displaying the resulting `RepositoryInsights` object. To choose a repository to analyse, the user has to insert the repository identifier (ID or full name) and the platform (`github.com` or `gitlab.com`). Optionally, the user can provide the specific commit hash (e.g. `13157a635cadda957d5d35f7b08a0fa902aa04af`) for the analysis. If this is not specified, then the most recent commit of the default branch will be used.

Once the UI has received the results from the `Analysers Manager`, it has to process them for a proper visualization. We have already seen that the logic for mer-

[30] the number of `PracticeCategory` objects instantiated is $N_{detectors} * N_{workflowfiles}$

ging `RepositoryInsights` objects, shown in Listing 5.1, does not merge related `PracticeCatetory` objects. To avoid showing multiple `PracticeCategory` instances for the same category, a key component of the User Interface is the merging logic for this class.

The operations to combine `PracticeCategory` objects are:

- Grouping `PracticeCategory` instances of the same category.
- Summing up the `elementsFound` fields.
- Appending all the `references` lists.
- Combining the `properties` field, merging singularly key by key. This logic is not trivial, as the `Map` `properties` can have anything as a value. For this field, we group by practice (or pattern) name and combine the `CodeReference` lists inside the `ProperyWithReferences` objects.

Listing 5.10 shows an example of implementation, while Figure 5.7 shows an example of two instances merged.

```
def combine_same_practice_category(practices_category: List[PracticeCategory]) ->
    PracticeCategory:
    return PracticeCategory(
        name=practices_category[0].name,
        elements_found =
            sum(pc.elements_found for pc in practices_category),
        references=
            [ref for pc in practices_category for ref in pc.references],
        properties=
            combine_properties([pc.properties for pc in practices_category])
    )
```

Listing 5.10: Python code for merging `PracticeCategory` objects. The real code in the User Interface is written in Typescript. It has been transcribed here to Python for higher understandability. The `combine_properties` method has not been included.

Figures 5.8 and 5.9 show how the insights are displayed in the UI.

5.5. Testing

Besides evaluating the insights produced by our framework, as presented in Chapter 6, we have also implemented more than 50 between unit and integration tests. This section briefly describes what we have tested for the Java and Python applications. The Angular Application doesn't present any test as the code inside is trivial.
Note that we did not implement any test that runs the framework as a whole, but we focused on testing the Python and Java applications independently.
Overall, these tests improve the quality of our implementation and add confidence that the framework we propose functions as it should.

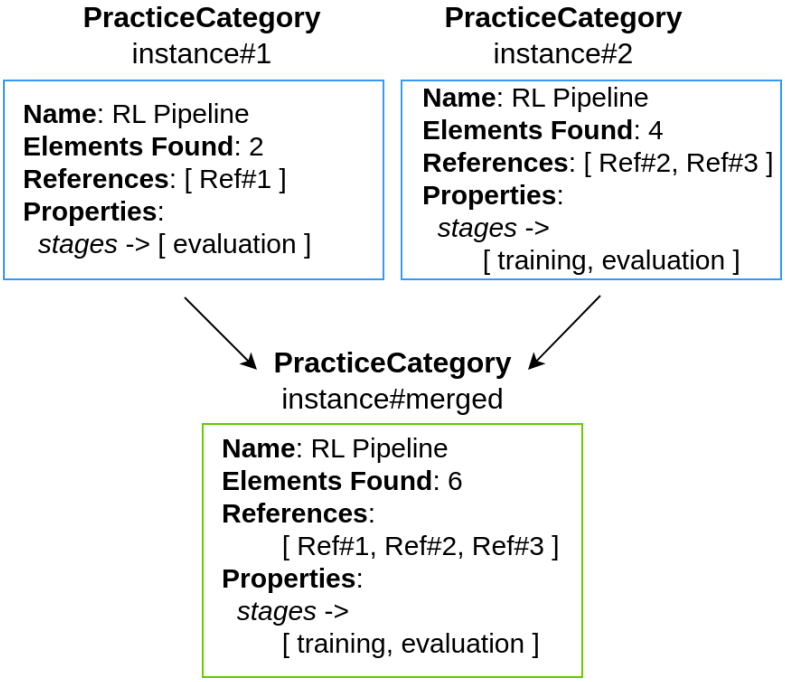

Figure 5.7.: Example of `PracticeCategory` objects being merged. Note that the *evaluation* stage is not duplicated in the merged instance.

5.5.1. Python Application

The Python Application contains 36 tests in total, both unit and integration tests. All the unit tests focus on the `extraction` package[31] and ensure the correctness of the *ASTWrapper* classes. Specifically, they all check that the values returned from these classes (e.g. a function name) are the expected ones extracted from mock Python files. These tests do not access the line numbers as these come from the `ast` library and are *expected* to be correct.
The integration tests check the `search` and `detection` packages, ensuring that all the `Detector` classes produce the expected insights when analysing mock `.py` files.

5.5.2. Java Application

Inside the Java Application, we can find 20 unit tests performed mostly on two components:

- The `extraction` package[32], to ensure all the parsing logic works correctly. These tests compare both the object values (e.g. the job name) and the line

[31] Check Figure 5.4 for more information.
[32] Check Figure 5.6 for more information.

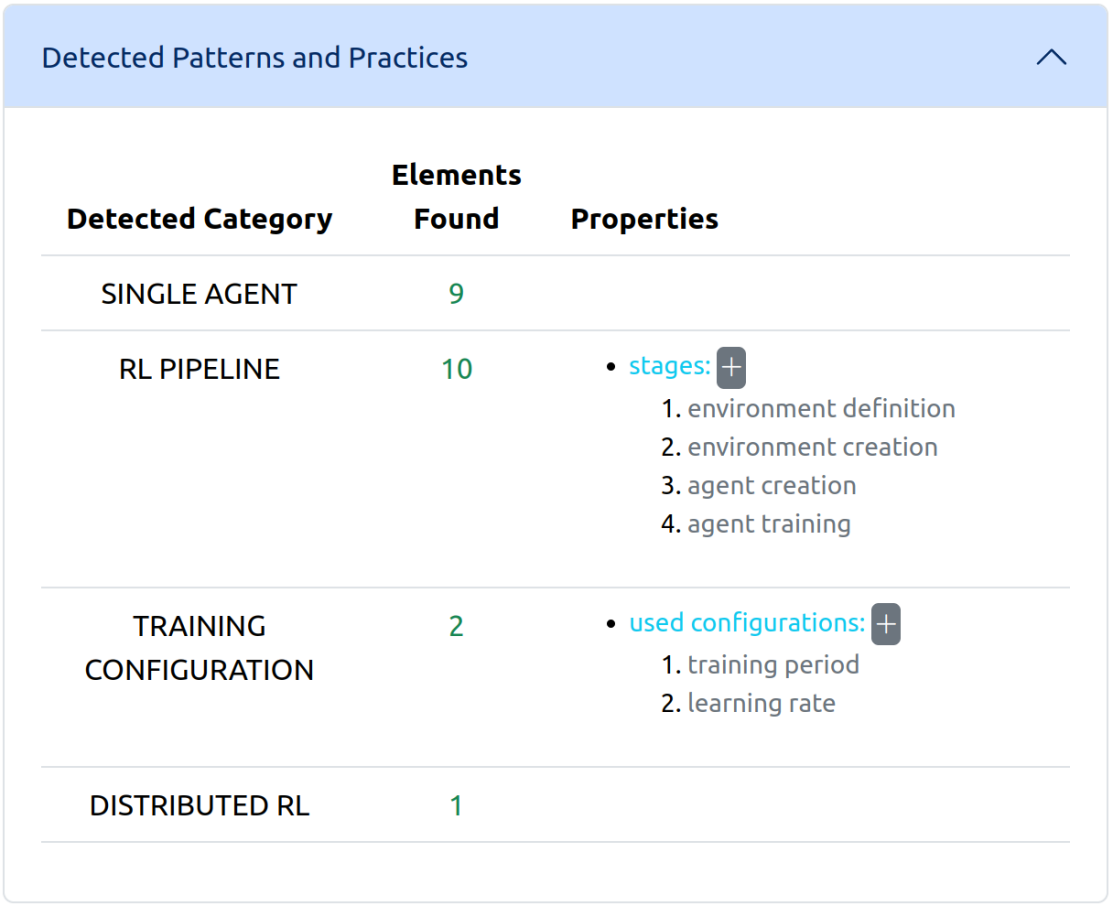

Figure 5.8.: Example of insights visualised with UI. Four pattern and practice categories have been identified for the repository analysed. The `properties` column lists the specific patterns and practices identified, for the categories with more than one.

numbers parsing (e.g. the job clause goes from line X to line Y) with the expected ones. To perform these tests, example workflow files are employed.

- The `fetching` package, to check the logic for downloading GitHub and GitLab repositories.

File: example3.py

```
2: import gym

7: class MazeGameEnv(gym.Env):
8:     def __init__(self, maze):
9:         super(MazeGameEnv, self).__init__()
10:         self.maze = np.array(maze)  # Maze represented as a 2D numpy array
11:         self.start_pos = np.where(self.maze == 'S')  # Starting position
12:         self.goal_pos = np.where(self.maze == 'G')  # Goal position
```

Figure 5.9.: Example of code references visualised in the UI. The lines with `isDecisive = true` are highlighted in green, while the rest are grey. Note that the content of some lines is cut in the image

6. Evaluation

This chapter describes how we evaluated our framework and the results obtained using the metrics precision, recall, and F1-score over the 15 case studies.

6.1. Design

To evaluate the framework proposed in this thesis, we have constructed a dataset of 15 case studies and manually labelled, for each file inside the systems, the lines of code (i.e. line numbers) relevant to each pattern and practice investigated in Chapter 4. All these line numbers have been saved in `json` files, one per each category of patterns and practices. An example file can be seen in Listing 6.1.

```
{
  "training config : gamma": {
    "CS1":{},
    "CS2":{},
    "CS3": {
      "example2.py": [18, 47]
    },
    "CS4": {
      "example2.py": [87]
    },
    "CS5": {}
    "CS6":{},
    "CS7":{
      "configPy.py": [19]
    },
    "CS8":{},
    "CS9":{},
    "CS10":{
      "train_hier.py": [186],
      "train_hetero.py": [216]
    },
    "CS11":{},
    "CS12":{
      "masac/masac.py": [55, 294],
```

```
25        "masac/masac_jax.py": [52, 369]
26      }
27    }
28  }
```

Listing 6.1: Example of labelled line numbers for one best practice of the *Training Configuration* category. To minimise the size of the labelled data, only the files with one or more relevant lines have been included.

Using two scripts (one for the Python Application and a second one for the Java application), we iterate through every `Detector` and every relevant[1] files. Each file is singularly analysed by a `Detector` and the lines to be evaluated are extracted from the returned `PracticeCategory` object[2].
To measure the accuracy of our framework, we process the True Positives (TP), False Positives (FP) and False Negatives (FN) produced among these lines. More in detail:

- The True Positives are the line numbers relevant to a pattern or practice, which have been correctly identified as relevant.
- The False Positives are the line numbers *not* relevant to a pattern or practice, which have been wrongly identified as relevant.
- The False Negatives are the line numbers relevant to a pattern or practice, which have *not* been identified as relevant.

The script for calculating these figures for the Python Analyser is shown in Listing 6.2. Using TP, FP and FN, we can calculate three metrics for accuracy:

$$\text{Precision} = \frac{\text{TP}}{\text{TP+FP}}\,;\text{Recall} = \frac{\text{TP}}{\text{TP+FN}}\,;\text{F1-score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{6.1}$$

Note that, inside the evaluation scripts, we have included the `verbose` option. When set to true, all the True Positives, False Positives and False Negatives are printed to console. This was done to best explain the final metrics obtained by our framework.

```
1 def process_detector_results(detector, practice_id):
2   results = {}
3   for system_id, system_directory in CASE_STUDIES.items():
4     system_results = {}
5     system_py_files =
6     fileIO.get_all_python_files_in_subfolders(system_id)
7
```

[1] The `.py` files for the Python Analyser, and the workflows for the Workflow Analyser.
[2] Note that only the lines with `isDecisive = true` are considered. As outlined in Section 5.2, the detectors add many lines with `isDecisive = false` in the result, to provide more context to the user.

```
    for python_file in system_py_files:
      try:
        practice_category =
        detector.analyse_file_string(fileIO.read_file(python_file))
        # compare results with json file
        system_results[python_file_relative_path] =
          compare_lines(practice_category, practice_id)
      except Exception as e:
        system_results[python_file_relative_path] = []

    results[system_id] = system_results

  return results
```

Listing 6.2: Script for the evaluation of the Python Analyser. The logic to get all the TP, FP, and FN is performed inside the `compare_lines` method, omitted here.

6.2. Dataset Construction

To understand the performance of the application we propose, we constructed a dataset containing 15 case studies, of which 10 are open-source systems gathered from GitHub and 5 are tutorial systems. Table 6.1 displays all the case studies. In this section, we describe the process we used to build such a dataset.

6.2.1. Mock Systems

To test the insights produced by our application, we have started by including 4 mock case studies using tutorial source code. For each analysed RL library (i.e. `RLlib`, `SB3`, `gym`, `PettingZoo`), we have added three example scripts obtained from the libraries' documentation.
Then, we gathered open-source systems, as described in the following subsection. Given that, among these case studies, we didn't find any RL system applying continuous training[3] [15], we have chosen to create an additional case study (CS5) applying this practice, relying on one *stable-baselines3* tutorial.

6.2.2. Open-source Systems

To gather open-source systems, we have relied on GitHub's global code search[4]. Starting from 200M[5][6] public repositories, we have filtered files using *signatures* of

[3] CT is an MLOps practice, but we have included it inside the RL pipeline.
[4] `https://github.com/search/advanced`: accessed September 2024
[5] `https://github.com/search?q=is%3Apublic&type=repositories`: accessed September 2024
[6] The actual number may be smaller as some repositories are not always indexed in the code search: `https://github.com/orgs/community/discussions/107482`

ID	Fullname	Considered Hash	Description
CS1	mock_systems/ rllib	b8d20116d7a0d9b372a6 22aa57a63b8f6e5aa3b7	Three tutorial scripts from RLlib.
CS2	mock_systems/ stable-baselines3	83656fb37346acb99fe5 da5e3eaaeebf7191ff95	Three tutorial scripts from stable-baselines3.
CS3	mock_systems/ gym	119e58e4e204568c2732 82969bc399ed0f5aa860	Three tutorial scripts from gym/gymnasium.
CS4	mock_systems/ pettingzoo	da3b919d180f31a2ac7d 60972db17a4cc288f1b2	Three tutorial scripts from PettingZoo.
CS5	mock_systems/ rl-pipeline	c8c2a861c1650286e5b9 b48c9ecf4d5fe350b118	Example system applying continuous training on an RL agent.
CS6	drkostas/ Minecraft-AI	747cda1854388ffe3733 57d1f217b95c7710b903	Trains an agent to solve missions on Minecraft.
CS7	AhmetFurkanDEMIR/ SuperMarioBrosRL	297fdaf6d2c53a802c0c 2f85cc86289dc8db2063	Trains an agent to run over an environment themed as Super Mario.
CS8	tsmatz/ minecraft-rl-example	b4712b6c30d14fe830fd bd1ac062ddcf39896028	Trains an agent to run over an environment themed Minecraft.
CS9	chunky/ sb3_to_coral	f5f71cd36768e33c8c13 f28ea8aed7fbf8ec6f7d	Trains SB3 agents to run over Gym environments and exports them in multiple formats.
CS10	IDSIA/ hhmarl_2D	1d6a1fd6e794b733c3b0 218a0b6d6fa607a1bf50	Trains MARL agents over custom environments to learn aeroplane combat techniques.
CS11	CN-UPB/ NFVdeep	d1ee9e5bdb28bc9bad33 9863a180cb9d4a5c7cb7	Reproduction of a deep reinforcement learning paper.
CS12	ffelten/ MASAC	a517bcd489447e074dfa b6bb2d5b9a257225f004	Trains custom agents to run over PettingZoo environments.
CS13	adin786/ energy_forecast	1f481bf654a0f7d23ba2 e80f18c94e0949b13706	Forecasts energy consumption in the UK.
CS14	rakesh-racharla/ mlopsdemo	bb544862df7fffb6a729 96a7afb2ffb70fc69b73	Demo MLOps system which trains neural networks over toy data.
CS15	ChiefPatrik/ mBikePredictions	892e463b48ff4668ffcc 8c156c7873f21e049f2a	Forecasts weather.

Table 6.1.: Description of the case studies used for the evaluation. Note that case studies from CS1 to CS5 can be found under `https://gitlab.com/$FULLNAME`, CS6 to CS15 under `https://github.com/$FULLNAME` and that the commit hash is split into two lines for better readability. The commit hash has been included to inform about the exact version of each repository that we used for the evaluation.

RL and MLOps technologies. It must be noted that GitHub's global code search only returns 100 results, sorted descending by recency[7] (most recent first). A filter based on recency is often used for building repository datasets [51, 12]. More in detail, as depicted by Figure 6.1, we have followed two different approaches for RL and MLOps results:

- For RL technologies:
  - We have looked for signatures such as `from gym import`, restricting results to `.py` files. We have run these search queries for RLlib, SB3, gym, PettingZoo, gathering 400 results (out of 700k signaled by the GitHub UI).
  - We have removed all repositories with less than 15 stars and more than 4000 LOC in `.py` files, ending up with 18. This step required short mining scripts, to check the number of Python lines of code.
  - We have removed, by hand, all false positives. Examples of these were libraries only present in `requirements.txt` or comments. At the end of this process, we ended up with 7 case studies.
- For MLOps technologies:
  - We have looked for signatures of DVC[8] usage, such as the `setup-dvc` GitHub action[9] inside the `.github/workflows` folder, which resulted in 100 results (448 in total written by the UI).
  - We have looked for strings such as *docker*, *image*, *push* and *artifact*. From this step, the number of repositories went down to 25.
  - We have removed, by hand, all false positives. Many of these had workflows executing docker operations, but not docker push. The final result is 3 case studies.

Given the high number of repositories found for RL technologies, we have filtered them based on popularity (i.e. number of stars) as this metric can indicate the repository's quality [51]. This was not done for MLOps technologies given that, with popularity filters, we hadn't found any repository. Nevertheless, this is because we have looked for only one MLOps technology (DVC) and four RL ones.

In addition, given that we had to manually label each line of code for the evaluation, we have filtered the GitHub repositories resulting from RL technologies based on the number of lines of Python code.

[7] `https://github.com/orgs/community/discussions/66434`: accessed September 2024
[8] A Data Versioning technology popular for MLOps projects.
[9] `https://github.com/iterative/setup-dvc`: accessed September 2024

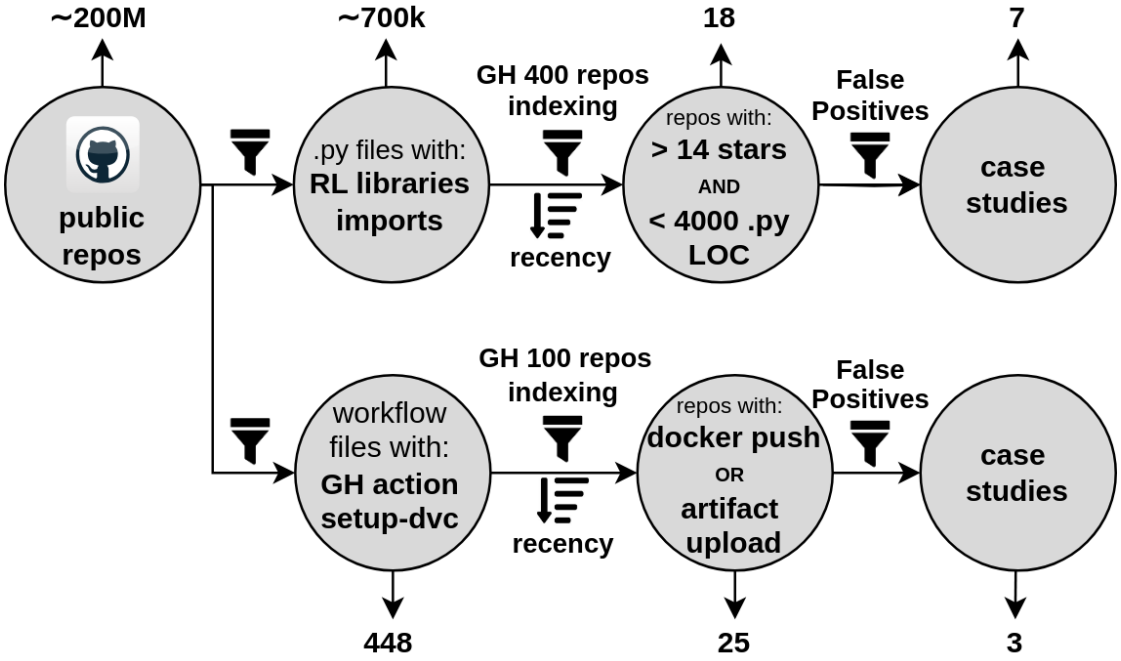

Figure 6.1.: Steps applied to build the dataset of open-source case studies.

6.3. Results

This section discusses the results summarised in Tables 6.2, 6.4, 6.3. All these tables show evaluation metrics. In detail: Table 6.2 is related to the extracted line numbers over the 9 patterns and practices categories, Table 6.4 to extracted line numbers for every single pattern and practice[10], and Table 6.4 to characteristics extracted from the source code (only for the Workflow Automation category).
In the following subsections, we describe each category's metrics. Overall, our framework has a combined F1 score of 0.93 for the extracted line numbers. More in detail, the score is 0.99 for the MLOps patterns and practices and 0.82 for the RL ones.
Furthermore, the precision is higher than the recall for all categories but one (workflow automation). This is the direct result of two characteristics of our approach:

1. Both in the Workflow Analyser and Python Analyser, we parse the code into logical elements (e.g. a function call, a workflow step), which are then analysed to look for specific signatures. This parsing effort increases the likelihood of an element being correctly identified. Nevertheless, given that not all elements are parsed (e.g. `for` loops in the Python Analyser), many elements can be false

[10] For the categories with only one pattern or practice (e.g. Distributed RL), the metrics are not included.

Category	True Positives	False Positives	False Negatives	Precision	Recall	F1-score
Checkpoint	36	3	33	**0.92**	0.52	0.67
Hyperparameter Tuning	9	0	0	**1.00**	**1.00**	**1.00**
Distributed RL	53	2	12	**0.96**	0.82	0.88
Single-Agent RL	114	15	32	0.88	0.78	0.83
Multi-Agent RL	63	4	46	**0.94**	0.58	0.72
RL Pipeline	246	26	47	**0.90**	0.84	0.87
Training Configuration	88	19	34	0.82	0.72	0.77
RL Categories (combined)	609	69	204	**0.90**	0.75	0.82
Workflow Automation	1255	16	0	**0.99**	**1.00**	**0.99**
Workflow Results	8	1	1	0.89	0.89	0.89
MLOps Categories (combined)	1263	17	1	**0.99**	**1.00**	**0.99**
All Categories (combined)	1843	83	187	**0.96**	**0.91**	**0.93**

Table 6.2.: Metrics for the extracted line numbers over each investigated category and with the categories combined.

negatives.

2. We rely more on technology-based elements than heuristic-based ones. As explained in Section 5.1, this leads to higher precision but lower recall.

6.3.1. Reinforcement Learning Categories Results

Table 6.2 shows high accuracy in identifying the code line numbers with 0.90 precision, 0.75 recall and 0.82 F1 score. Below we discuss the results in more detail for each RL category, providing examples of false positives and negatives.

Checkpoint

The precision of the checkpoint category is very high (i.e. >= 0.90), but overall,

Pattern or Practice	True Positives	False Positives	False Negatives	Precision	Recall	F1-score
Checkpoint	21	1	17	**0.95**	0.55	0.70
Checkpoint Path	11	2	12	0.85	0.48	0.61
Checkpoint Frequency	4	0	4	**1.00**	0.50	0.67
MARL	49	4	32	**0.92**	0.60	0.73
Fully Coop. MARL	4	0	0	**1.00**	**1.00**	**1.00**
Fully Comp. MARL	8	0	4	**1.00**	0.67	0.80
Mixed Coop.-Comp. MARL	2	0	10	**1.00**	0.17	0.29
Environment Definition	32	0	2	**1.00**	**0.94**	**0.97**
Environment Creation	97	4	5	**0.96**	**0.95**	**0.96**
Agent Creation	55	2	23	**0.96**	0.71	0.81
Agent Training	26	2	8	**0.93**	0.76	0.84
Agent Save	11	2	4	0.85	0.73	0.79
Agent Evaluation	25	16	5	0.61	0.83	0.70
Seeding	46	8	3	0.85	**0.94**	0.89
Learning Rate Usage	16	1	1	**0.94**	**0.94**	**0.94**
Gamma Usage	6	0	4	**1.00**	0.60	0.75
Training Period Configuration	20	10	26	0.67	0.43	0.53
Workflow Stages	3	1	0	0.75	**1.00**	0.86
Workflow Jobs	1240	13	0	**0.99**	**1.00**	**0.99**
Jobs Dependencies	12	2	0	0.86	**1.00**	**0.92**
Job Artifacts	3	1	0	0.75	**1.00**	0.86
Docker Images	5	0	1	**1.00**	0.83	**0.91**

Table 6.3.: Metrics for the extracted line numbers over each pattern and practice.

Pattern or Practice	True Positives	False Positives	False Negatives	Precision	Recall	F1-score
Workflow Stages	1	0	0	**1.00**	**1.00**	**1.00**
Workflow Jobs	24	0	0	**1.00**	**1.00**	**1.00**
Jobs Dependencies	24	0	0	**1.00**	**1.00**	**1.00**

Table 6.4.: Metrics for the extracted characteristics for the Workflow Automation category.

the F1 score is lower, 0.67, because of many false negatives. More in detail, Table 6.3 shows similar tendencies for the metrics of all the patterns and practices in the checkpoint category: high (i.e. between 0.75 and 0.89) precision, low (i.e. < 0.60) recall and moderate (i.e. between 0.60 and 0.74) F1 score.
Examples of false positives are library functions we do not detect: *torch.save*, *orbax.checkpoint.PyTreeCheckpointer*. For the checkpoint frequency pattern, we have found four instances where it is applied with a modulo operation (e.g. *if iteration % 100*). We do not parse the `if` statement; therefore, these lines were not identified.

Hyperparameter Tuning

The detection of hyperparameter tuning results in all perfect (i.e. equals to 1.0) metrics. This is because few case studies apply the practice, and all these studies use `ray.tune`.

Distributed RL

The results for the distributed RL practice show high values for the metrics. Given that most of the relevant lines are the initialisation and configuration of a distributed agent, these can be easily identified using the function/method names from libraries. While we have found a few custom algorithm implementations across the case studies, these did not implement a distributed pattern and were therefore not considered in the ground truth.
Examples of false negatives are custom configuration lines (e.g. a dictionary defining the number of workers) and old APIs, which were not present anymore in the version of RLlib we analysed (e.g. *DQNTrainer* class) and that, for this reason, we have not included in the among the `ElementQuery` objects.

Single-Agent RL

Table 6.2 shows high values for all metrics of the SARL category. The combination of technology-based and heuristic-based elements works well for Single-Agent RL. While the presence of library elements is high, many lines can be identified only with heuristics (e.g., custom environment or agent instantiations).
Compared to the other practices, there is a considerable amount of false positives, which results in a precision lower than 0.90. This is caused, among other things, by MARL agents and environments being misidentified as SARL.
Examples of false negatives are SARL agents and environments completely missed (e.g. *tune_env = deepcopy(env)*.

Multi-Agent RL

The results for the MARL category show high precision, low recall, and moderate F1 score. As Table 6.2 shows, there are many false negatives (compared to the number of true positives). Many of these false negatives are custom configurations. As an example, 8 of them are used in the file *config.py* from CS10, such as *friendly_kill* (whether two cooperating agents can kill each other) or *glob_frac* (fraction of reward sharing).
In addition, other false negatives are caused by MARL agents and environments being ignored because they are identified as SARL, as anticipated in the previous paragraph.
Nevertheless, despite these shortcomings, the F1 score is still adequate, at 0.72. This is thanks to the extensive technology-based and heuristic-based elements we have included in the detector.

Besides extracting lines of code for the MARL pattern, we also investigated the types of environments used. Table 6.3 displays a considerable change in performance for the three environment types. Our framework results in an F1 score of 1.00 for fully cooperative environments, 0.80 for fully competitive and 0.29 for mixed cooperative-competitive. The big difference between these metrics is partially caused by the small number of lines of code found in the 15 case studies. Nonetheless, the poor performance is also due to the complexity of identifying these specific patterns for custom MARL environments. Determining whether an environment is cooperative, competitive or mixed is simple for library environments as it is enough to manually label each of these. On the other hand, the task is harder for custom environments because it requires detecting specific elements inside *a class* (e.g. an assignment to *opponent*) and not inside *a file*. The design of our `Detector` class revolves fully around detection on the file level and not the class level, making this task hardly doable.

RL Pipeline

Table 6.2 displays the combined metrics for the RL pipeline category. These values are all either high or very high. Table 6.3 shows the performance reached by our framework in each specific pattern and practice. The environment definition and creation are the most straightforward patterns to identify, mainly thanks to practitioners' *habit* of ending environment names with *Env*.
The metrics on three of the other patterns and practices (i.e. agent creation, training and save) are also high. For the agent creation, the detection process resulted in many false positives. Examples of these are library classes and functions not analysed in this thesis (e.g. *onxx.load*) and elements we did not find when analysing libraries' documentations (e.g. *ray.rllib.policy.Policy*).
Conversely, identifying the agent evaluation results in many false positives, even though the F1 score is not low. Some heuristic-based elements we use (e.g. any method/function call with *eval* inside the name) identify evaluation configurations[11], causing lower precision.

We can make a final observation of RL Pipeline results. Even though the relevant lines from an environment definition, environment creation and agent creation are also included in the SARL and MARL practices, these two have lower metrics. This is expected because, for SARL and MARL, the detection process has to identify these steps and distinguish whether the step was related to a single agent or multiple ones.

Training Configuration

The combined metrics (see Table 6.2) for the training configuration category are good: a high precision and F1 score and an adequate recall. More specifically, Table 6.2 shows that the metrics for seeding, learning rate and gamma are all either high or very high, with one exception (recall of gamma). This is easily explainable as the elements used across the case studies are homogeneous (e.g., a seed initialisation is usually done on a variable with *seed* in the name).
Conversely, we have found identifying the training period to be more challenging. This pattern results in the second-lowest F1 score in this thesis: 0.53. Nevertheless, compared to the lowest (Mixed Cooperative-Competitive MARL), many relevant lines exist for the training period.
By looking at the false negatives that we have found, we can mostly see *implicit* training periods (e.g. *for i in range(100)*) and instances with custom names (e.g. *target_steps*). Identifying the former would require adding more Python elements to be parsed (i.e. *for* loops), while the latter would require defining effort in defining

[11] As specified in Section 5.1, we are only interested in lines where the evaluation takes place and not where configurations are created.

heuristics.

6.3.2. MLOps Categories Results

Using Tables 6.2 and 6.4 we can observe that producing insights for the 2 MLOps categories of patterns and practices is more straightforward than the other RL categories. In the following paragraphs, we discuss the two categories singularly.

Workflow Automation

The metrics for the detection of the workflow automation category are the sum of three identified patterns and practices (i.e. workflow stages, workflow jobs and jobs' dependencies), whose individual results are displayed in Tables 6.3 and 6.4. Overall, we can observe two key things:

- The metrics are all either high or very high.
- For the detected lines, contrary to all RL practices, there are more false positives than false negatives.

The first tendency is the direct effect of the very simple *syntax* of GH/GL workflow files, compared to the Python programming language. Identifying relevant lines for stages, jobs, and jobs' dependencies is trivial, and it is enough to parse the simple structure of YAML workflow files. This is the same for the extraction of characteristics (e.g. job dependencies)
The second tendency is caused by one of the two parsers we use (SnakeYAML), which considers empty lines part of a YAML clause (e.g., the lines between two defined jobs). We did not consider these empty lines in the ground truths; therefore, they are false positives. There are no other false positives.

Workflow Results

The identification of workflow results, as introduced in Section 5.1.9, involves detecting two different patterns: producing job artifacts and docker images. The former is trivially identifiable by parsing the YAML clauses. As for the Workflow Automation practice, the single false positive is caused by an empty line. The latter can be identified by proper YAML parsing and by analysing the commands run in a workflow. The false negative is a GitHub action (i.e. *jwalton/gh-ecr-push*), which pushes docker images to Amazon ECR instances. We did not incur this action during the research conducted for this thesis.
It's still important to note that the number of found occurrences of both patterns is low, which would require adding more case studies.

6.4. Professional Usage

The framework proposed in this thesis will be extensively used in a large software company to remain anonymous, which we will denote as Y. At the current stage, our framework is being merged with an application (which we will denote as A_Y) already developed inside Y, which focuses on analysing GitLab repositories. After a successful merging process, the framework will be deployed internally in Y to analyse hundreds of projects.
In this section, we briefly describe the main challenges we have encountered, the modifications we have made to improve the quality of the framework, and the lessons learned.

6.4.1. Integration Challenges

Before describing the issues faced up to this point during the integration phase, we must briefly summarise A_Y. A_Y is a full-stack application with one backend container (written in Java), one frontend container (written in Angular), one containerised database (a MongoDB instance) and multiple volumes. The backend and frontend containers are deployed using Kubernetes[12] while the database is deployed on Amazon S3[13]. Similarly to our framework, a user inserts a repository identified in the front end and chooses the analysis type (multiple analyses are performed in Y). The backend then fetches the selected repository and starts the analysis process. After the analysis is done, the user can visualise the results in the UI.

From our brief description, it can be seen that our architecture is pretty similar to A_Y's. The same is true for the use cases.
To merge the applications, we have chosen to combine[14] the two Java codebases, the two Angular codebases and add a Python container to A_Y architecture. These design decisions have been chosen among multiple options. For example, two containers could have been used for the two Java applications to limit the refactoring process.

Up to this point, we have faced several challenges during the integration effort:

- *Different technology versions.* Both the Java applications and the Angular applications used some of the same libraries (e.g. Spring Boot[15], Bootstrap [16]) but with different versions. We have, therefore, changed the versions in our proposed framework to match the ones inside A_Y. This challenge

[12] `https://kubernetes.io/`: accessed September 2024
[13] `https://aws.amazon.com/pm/serv-s3/`: accessed September 2024
[14] The source code combination also includes rewriting part of the codebases.
[15] `https://spring.io/projects/spring-boot`: accessed September 2024
[16] `https://getbootstrap.com/`: accessed September 2024

required refactoring some classes and functions as the APIs used didn't exist (for downgrades) or were deprecated (for upgrades).

- *Different Conventions Applied.* In A_Y, we found different styles of writing classes. For example, A_Y used many smaller classes compared to fewer bigger ones. In addition, A_Y greatly used utility libraries to reduce the number of lines of code per class.
  Furthermore, even though both applications used a layered architecture, this was applied differently. These discrepancies resulted in more effort during the refactoring process. Nevertheless, this issue is expected among any two codebases.

- *Overlapping Logic.* Both applications must fetch repositories given an identifier and automatically return analysis objects. During the integration effort, we have spent considerable time removing overlapping logic and adapting our application to use A_Y classes.

- *Different Patterns Applied.* Besides lower-level design patterns, the two Java applications offered different API architectures. In our proposed framework, all the communications are synchronous as they are simpler to implement and test. A_Y used asynchronous communication for many operations, among which we can find the analysis of a repository, as this improves efficiency. This paradigm shift required considerable time to rewrite our application's internal working and adapt the synchronous communications into asynchronous ones.

6.4.2. Lessons Learned

The integration process completed so far has taught us multiple valuable lessons. We can divide these into three main categories:

- *New Technologies Learned.* While adapting our application's conventions and patterns, we extensively used the libraries applied inside A_Y. This effort is a valuable experience using such technologies in a mature product, and it opened us to using them in future projects.

- *New Programming Styles Learned.* In A_Y, we have found many libraries whose goal was reducing the number of lines of code inside a class. Not only has this introduced these technologies, but we also increased productivity and improved coding experience by reducing boilerplate code.

- *New Development Process Learned.* During the collaboration with Y, we have worked alongside Y's employees, and we have experienced the development process found inside a big software company such as Y.

6.5. Replication Package

The replication package is `.zip` file, found at the faculty GitLab. It contains two things:

- The `docker-compose.yml` deployment file, which allows to locally run the framework and access it at `localhost:4242` with a browser.
- The evaluation scripts, which allow running the evaluation for each Practice Category locally.

The replication package contains all the relevant commands and information to properly replicate this thesis in the `README` file.

7. Discussion

This chapter discusses our findings and answers the three research questions addressed by this thesis.

RQ1

From a scientific literature review and 23 practitioner sources, we identified 9 categories of patterns and practices inside Reinforcement Learning and MLOps architectures, for a total of 25 between patterns and practices.
These categories often overlap. For example, the RL Pipeline category contains the *Environment Definition* and *Environment Creation* patterns, which are also related to the Single-Agent and Multi-Agent categories. The same is true for the *Agent Creation* pattern. In addition, *Distributed RL* overlaps with the *Agent Creation* pattern.
Furthermore, some patterns and practices can be considered specific cases of others: the *Hyperparameter Tuning* best practice is a particular scenario of *Agent Training*.

RQ2

The extraction of relevant line numbers performs well, as can be seen from Table 6.2, with an F1 score of 0.96 for all categories combined, 0.99 for MLOps categories, and 0.90 for the RL ones. While specific metrics vary from category to category, there is only one (Multi-Agent RL) with a non-high (i.e. < 0.75) F1 score, and all the others result in a high or very high F1 score. For certain patterns and practices (e.g. hyperparameter tuning), the number of usages across the analysed case studies is low. Furthermore, the precision is high for all categories while the recall is non-high for three of them. Additional effort is required in future research to find more case studies and to improve the recall.

For the second type of insight (i.e. displaying elements found in the source code), the metrics in Table 6.4 show a perfect score for each practice. Nevertheless, this insight is limited to one MLOps category and should be extended to the others in future research.

RQ3

When a detector has been coded, the detection process is entirely automatic. Nevertheless, defining which source code elements are to be identified still requires manual

effort. In this work, we have used technology-specific and heuristic-based elements and applied our framework to case studies primarily employing the technologies we focused on. We can discuss the level of automation of our framework in three scenarios:

- **New Python libraries and workflow actions**. In these cases, reaching *adequate* detection metrics *may* require adding new technology-based elements by reading the related documentation. This may not be necessary since we have also defined heuristic-based elements.
  Overall, this process would be easier for new Python libraries as it is enough to change the `DetectorConfig json` files. Conversely, more effort is needed for workflow-related technology because small changes to the source code are needed. Nevertheless, for both cases, the effort is pretty low.

- **New programming languages**. Even though most RL programs are written in Python, analysing other programming languages (e.g. C++) may be desired. The effort required in this remote scenario would be much higher than for new Python libraries. The detection logic of the Python Analyser relies on *parsed* elements of source code. For this reason, analysing a new programming language requires creating classes equivalent to the `ASTWrapper` we use. Even though the effort is high, for our analysis, we only used 7 basic elements[1] present in most programming languages. Given the high performances shown by RQ2, the effort may be worth it.

- **New CI/CD technologies**. In this thesis, we have analysed MLOps patterns and practices, focusing on two CI/CD technologies: GitHub and GitLab workflows. These are not the only CD/CD tools used by practitioners. Nevertheless, most of these tools use the same concepts (i.e. jobs, steps, etc.) and allow writing workflows using YAML files. Adapting our framework to new CI/CD technologies requires new parsing logic. Contrary to programming languages, this effort is not high, and we have proved this by manually parsing GitHub/GitLab workflows in this thesis, combining the capabilities of Jackson and SnakeYAML. By only parsing a handful of YAML clauses, it's possible to analyse most of a workflow's logic.

It's important to note that if the second and third scenarios required new analyser services developed in additional programming languages (besides Python and Java), these could be easily integrated with our framework, thanks to its distributed design.

[1] As it can be seen from Table 5.2, these elements are: assignments, calls, classes definitions, functions and methods definitions, imports, keywords, string constants

8. Related Work

In this chapter, we discuss work related to this thesis. Two sections are dedicated to RL and MLOps patterns and practices and their analysis, one section to similar works in source code analysis, and the last section to datasets used for source code analysis.

8.1. Reinforcement Learning Patterns and Practices

Research in Reinforcement Learning has already been conducted to list patterns and practices. Ntentos et al. [54] have identified 6 design decisions resulting in 29 possible design options, relying in their study on *grey literature*. Some of the patterns and practices listed by Ntentos et al. are present in this thesis.
Other studies have focused their research on a specific pattern. Samsani et al. [59] concentrated on *distributed reinforcement learning*, multiple authors [11, 68, 13, 55, 29] described the *Multi-Agent* pattern. Busoniu et al. [11] and Zhang et al. [68] have done general survey on MARL; Canese et al. [13] and Hernandez et al. [29] have focused on the challenges of the pattern; Oroojlooy et al. [55] have reviewed cooperative MARL.
These works have helped us answer RQ1 for RL patterns and practices. Nevertheless, none of these works attempts to detect the investigated practices from the source code. To our knowledge, no other study addresses RQ2 and RQ3 for RL patterns and practices.

8.2. MLOps Patterns and Practices

Similarly to Reinforcement Learning, numerous works in the MLOps field have addressed the need for collections of patterns and practices. Warnett et al. [66] have identified 7 architectural design decisions for MLOps systems. Their findings are based on *grey literature*. Besides listing design options, the researchers have provided multiple decision drivers (i.e. reasons) behind design decisions. Liang et al. [42] have focused on the importance and challenges of MLOps pipelines, discussing continuous training, deployment and monitoring of ML models.
These works have helped us answer RQ1. As for RL, no tools were offered to automatically detect the patterns and practices they reviewed.

Contrary to Reinforcement Learning, some works have investigated the usage of MLOps patterns and practices from source code. Calefato et al. [12] have examined 184 GitHub repositories to find how common ML-enabled workflows are. In addition, they have mined the workflows to find the most common actions run by these workflows. Nevertheless, the tool they implemented is not usable for automatically extracting insights from an architecture.
Like Calefato et al., Barrak et al. [9] have studied the adoption of MLOps in GitHub, explicitly concentrating on DVC[1], a data versioning technology. Their work analysed 391 GitHub repositories, answering how they evolved over time. Barrak et al.'s goal wasn't to automatically detect insights from source code.

8.3. Source Code Analysis

As previously described, to our knowledge, no study addresses the need for automatic detection of MLOps and RL patterns and practices. Nevertheless, this section presents works that have analysed source code from which we got inspiration.

To analyse Python source files, multiple studies [58, 14, 47, 56] employ the `ast` library. Robles et al. [58] determine Python proficiency based on the elements detected in the source code. Chen et al. [14] propose a new tool to identify *classic* code smells (e.g. Large Class) in Python programs. Mir et al. [47] and Pradel et al. [56] developed new frameworks to infer the type of variables and input parameters inside Python files, using ML. All these studies use `ast` to find elements they are interested in and extract from them the information needed. Using this information, they can run their analysis processes. This is similar to how we proceeded. In our case, we have built a layer on top of `ast`, which is usable via `json` configuration files.

For workflow analysis, we have employed a custom parser to analyse the files, internally relying on YAML parsers. This is due to the simplicity of GH and GL workflow APIs, which are not comparable to the rice syntax of most programming languages. Vassallo et al. [64] and Calefato et al. [12] have also made this design decision; the former to find CD (Continuous Deployment) smells in GitLab workflows, and the latter to mine tasks done inside GitHub workflows.

8.4. Dataset Construction

To see the performance (i.e. metrics such as accuracy, recall, etc.) of the developed tools and framework, it's necessary to test them on example projects. In the code analysis field, it's common for studies to limit the analysis to a single or few *mature*

[1] `https://dvc.org/`: accessed September 2024

projects for single-case or multi-case studies. Conversely, other works build a dataset of a *considerable* (e.g. more than one-hundred) number of open-source projects, usually in the form of public repositories scraped from the web, most commonly GitHub, GitLab and BitBucket.
Examples of single and multi-case studies are: Lenhard et al. [40] used several smell-detection tools on a single Java library containing 80 000 LOC (lines of code). Chen et al. [14] proposed a new smell detector for Python source code, testing it on five *big* (each with more than > 100 000 LOC) open-source project, such as `django`, `numpy` and `matplotlib`. Works using few *mature* systems have the advantage of testing their performance on complex case studies but may lack generalization. Each open-source project can have its conventions (e.g. for naming), and a tool with reasonable accuracy on one project may perform poorly on many other ones.

Other studies craft datasets of hundreds or more of *small* open-source projects to mitigate this issue. Vassallo et al. [64] developed a smell detector for pipeline configuration files and tested it on 5312 open-source projects found on GitLab. Calefato et al. [12] investigated MLOps practices among practitioners using 184 GitHub repositories. To scrape these repositories, they started from a dataset of 400k+ repositories [35] and filtered them, among other parameters, using keywords in the repositories' descriptions, number of stars, and recency of the last commit. Similarly, Barrak et al. [9] focused on the evolution of MLOps pipelines by analysing 391 GitHub repositories, using the GitHub public API and filtering for specific files.
Although using hundreds of open-source small to mid-size projects may improve the generalization of the study, it doesn't ensure that such systems are mature. For example, the datasets of Calefato et al. included many toy projects, either used to test a tool or for teaching purposes.

An evaluation phase conducted both on few *mature* codebases and many public open-source ones takes the best of both worlds. Nevertheless, this is often unfeasible, given the effort to build such datasets and evaluate the proposed tool on both repositories. For example, Pradel et al. [56] have such an evaluation phase. Their dataset construction was simplified by using closed-source Facebook projects to which the researchers had access.

In our study, we have scraped example systems from the web using GitHub's search API and filtering based on repositories stars, similarly to Barrak et al. [9] and Calefato et al. Given the nature of our evaluation process, which requires manual tagging of every relevant line of code, we employed only 15 case studies, contrary to the hundreds used by Calefato et al. and Barrak et al.

9. Threats to Validity

This chapter briefly discusses the validity threats to our work and how future research can address them.

To establish a robust *internal validity* of this study, we have carefully labelled the case study used in this thesis by hand. We have clearly defined which usages of practices we are interested in and have kept consistency across all the case studies during the labelling process. Finally, we have explored all the results obtained to best explain the final metrics.

In addition, we have implemented thorough tests across all components of the proposed application. Increasing the number of researchers working on the evaluation stage can reduce bias and errors and improve internal validity for future studies.

To improve the generalisation (i.e. *external validity*) of this thesis' findings, we have applied our application not only to tutorials but also to 10 open-source systems. The distinctiveness obtained from these case studies significantly improves external validity. Nevertheless, a greater dataset would help generalisation, especially for the patterns and practices with low usage across the 15 systems, such as hyperparameter tuning, workflow stages, and workflow results.

To address *construct validity*, we have investigated established RL and MLOps practices and relied on existing literature.

10. Conclusions

In this thesis, we have introduced a new framework for automatically detecting 9 categories of RL and MLOps patterns and practices using source code, relying on relevant source code elements initially identified by hand.
We believe that the effectiveness of our framework would assist practitioners in analysing new and large-scale systems, significantly reducing the effort needed for these processes. In addition, we have briefly described the integration of our framework inside a big software company, further motivating the need for our work.

The framework we have developed mostly relies on one Java and one Python application and makes heavy use of parsers. To display the results, one frontend application is used.
Employing 15 case studies, 10 of which are open-source systems, we have proved the accuracy of our proposed application, reaching a combined precision of 0.96, a recall of 0.91 and an F1 score of 0.93. More specifically, all categories but one (Multi-Agent RL) reach a high (i.e. >= 0.75) F1 score.
The framework can run fully automatic detection processes for the 7 technologies we have investigated. New technologies may initially require manual effort before being automatically analysed, depending on the specific type of technology.

In future research, our goal is to enhance the analysis capabilities of our tool to add more insights and additional patterns and practices, potentially reducing the manual effort required at the beginning for new technologies.

Bibliography

[1] Asynchronous proximal policy optimization (appo) for rllib. https://docs.ray.io/en/releases-2.34.0/rllib/rllib-algorithms.html#appo. Accessed: September 2024.

[2] How does an a3c work? https://medium.com/@dmonn/how-does-an-a3c-work-4e02266d1a96. Accessed: September 2024.

[3] Most popular tools - stack overflow survey 2024. https://survey.stackoverflow.co/2024/technology#most-popular-technologies-tools-tech. Accessed: September 2024.

[4] What are the most popular tools for reinforcement learning? https://www.linkedin.com/advice/0/what-most-popular-tools-reinforcement-learning-iuoie. Accessed: September 2024.

[5] Marcin Andrychowicz, Anton Raichuk, Piotr Stanczyk, Manu Orsini, Sertan Girgin, Raphaël Marinier, Léonard Hussenot, Matthieu Geist, Olivier Pietquin, Marcin Michalski, Sylvain Gelly, and Olivier Bachem. What matters in on-policy reinforcement learning? A large-scale empirical study. *CoRR*, abs/2006.05990, 2020.

[6] Marcin Andrychowicz, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, OpenAI Pieter Abbeel, and Wojciech Zaremba. Hindsight experience replay. *Advances in neural information processing systems*, 30, 2017.

[7] Noor Awad, Neeratyoy Mallik, and Frank Hutter. Dehb: Evolutionary hyperband for scalable, robust and efficient hyperparameter optimization. *arXiv preprint arXiv:2105.09821*, 2021.

[8] Bowen Baker, Ingmar Kanitscheider, Todor M. Markov, Yi Wu, Glenn Powell, Bob McGrew, and Igor Mordatch. Emergent tool use from multi-agent autocurricula. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020.

[9] Amine Barrak, Ellis E Eghan, and Bram Adams. On the co-evolution of ml pipelines and source code-empirical study of dvc projects. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 422–433. IEEE, 2021.

[10] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *CoRR*, abs/1606.01540, 2016.

[11] Lucian Busoniu, Robert Babuska, and Bart De Schutter. A comprehensive survey of multiagent reinforcement learning. *IEEE Trans. Syst. Man Cybern. Part C*, 38(2):156–172, 2008.

[12] Fabio Calefato, Filippo Lanubile, and Luigi Quaranta. A preliminary investigation of mlops practices in github. In *Proceedings of the 16th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 283–288, 2022.

[13] Lorenzo Canese, Gian Carlo Cardarilli, Luca Di Nunzio, Rocco Fazzolari, Daniele Giardino, Marco Re, and Sergio Spanò. Multi-agent reinforcement learning: A review of challenges and applications. *Applied Sciences*, 11(11):4948, 2021.

[14] Zhifei Chen, Lin Chen, Wanwangying Ma, and Baowen Xu. Detecting code smells in python programs. In *2016 international conference on Software Analysis, Testing and Evolution (SATE)*, pages 18–23. IEEE, 2016.

[15] Behrouz Derakhshan, Alireza Rezaei Mahdiraji, Tilmann Rabl, and Volker Markl. Continuous deployment of machine learning pipelines. In *EDBT*, pages 397–408, 2019.

[16] Paul M Duvall, Steve Matyas, and Andrew Glover. *Continuous integration: improving software quality and reducing risk*. Pearson Education, 2007.

[17] Christof Ebert, Gorka Gallardo, Josune Hernantes, and Nicolás Serrano. Devops. *IEEE Softw.*, 33(3):94–100, 2016.

[18] Theresa Eimer, Marius Lindauer, and Roberta Raileanu. Hyperparameters in reinforcement learning and how to tune them. In *International Conference on Machine Learning*, pages 9104–9149. PMLR, 2023.

[19] Lasse Espeholt, Hubert Soyer, Rémi Munos, Karen Simonyan, Volodymyr Mnih, Tom Ward, Yotam Doron, Vlad Firoiu, Tim Harley, Iain Dunning, Shane Legg, and Koray Kavukcuoglu. IMPALA: scalable distributed deep-rl with importance weighted actor-learner architectures. In Jennifer G. Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, volume 80 of *Proceedings of Machine Learning Research*, pages 1406–1415. PMLR, 2018.

[20] João Pedro Faustino, Daniel Adriano, Ricardo Amaro, Rúben Pereira, and Miguel Mira da Silva. Devops benefits: A systematic literature review. *Softw. Pract. Exp.*, 52(9):1905–1926, 2022.

[21] Dror G Feitelson, Eitan Frachtenberg, and Kent L Beck. Development and deployment at facebook. *IEEE Internet Computing*, 17(4):8–17, 2013.

[22] Francesca Arcelli Fontana and Marco Zanoni. A tool for design pattern detection and software architecture reconstruction. *Information sciences*, 181(7):1306–1324, 2011.

[23] Scott Fujimoto, Herke van Hoof, and David Meger. Addressing function approximation error in actor-critic methods. In Jennifer G. Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, volume 80 of *Proceedings of Machine Learning Research*, pages 1582–1591. PMLR, 2018.

[24] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[25] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In Jennifer G. Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, volume 80 of *Proceedings of Machine Learning Research*, pages 1856–1865. PMLR, 2018.

[26] Christian Haertel, Daniel Staegemann, Christian Daase, Matthias Pohl, Abdulrahman Nahhas, and Klaus Turowski. Mlops in data science projects: A review. In Jingrui He, Themis Palpanas, Xiaohua Hu, Alfredo Cuzzocrea, Dejing Dou, Dominik Slezak, Wei Wang, Aleksandra Gruca, Jerry Chun-Wei Lin, and Rakesh Agrawal, editors, *IEEE International Conference on Big Data, BigData 2023, Sorrento, Italy, December 15-18, 2023*, pages 2396–2404. IEEE, 2023.

[27] Danijar Hafner, Jurgis Pasukonis, Jimmy Ba, and Timothy P. Lillicrap. Mastering diverse domains through world models. *CoRR*, abs/2301.04104, 2023.

[28] Nicolas Heess, Dhruva TB, Srinivasan Sriram, Jay Lemmon, Josh Merel, Greg Wayne, Yuval Tassa, Tom Erez, Ziyu Wang, S. M. Ali Eslami, Martin A. Riedmiller, and David Silver. Emergence of locomotion behaviours in rich environments. *CoRR*, abs/1707.02286, 2017.

[29] Pablo Hernandez-Leal, Bilal Kartal, and Matthew E. Taylor. A survey and critique of multiagent deep reinforcement learning. *Auton. Agents Multi Agent Syst.*, 33(6):750–797, 2019.

[30] Dan Horgan, John Quan, David Budden, Gabriel Barth-Maron, Matteo Hessel, Hado van Hasselt, and David Silver. Distributed prioritized experience replay. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018.

[31] Jez Humble and David Farley. *Continuous Delivery Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley, 2010.

[32] Engin Ipek, Onur Mutlu, José F. Martínez, and Rich Caruana. Self-optimizing memory controllers: A reinforcement learning approach. In *35th International Symposium on Computer Architecture (ISCA 2008), June 21-25, 2008, Beijing, China*, pages 39–50. IEEE Computer Society, 2008.

[33] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. Analysis of large-scale multi-tenant GPU clusters for DNN training workloads. In Dahlia Malkhi and Dan Tsafrir, editors, *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*, pages 947–960. USENIX Association, 2019.

[34] Tatsuya Kasai, Hiroshi Tenmoto, and Akimoto Kamiya. Learning of communication codes in multi-agent reinforcement learning problem. In *2008 IEEE conference on soft computing in industrial applications*, pages 1–6. IEEE, 2008.

[35] Timothy Kinsman, Mairieli Santos Wessel, Marco Aurélio Gerosa, and Christoph Treude. How do software developers use github actions to automate their workflows? In *18th IEEE/ACM International Conference on Mining Software Repositories, MSR 2021, Madrid, Spain, May 17-19, 2021*, pages 420–431. IEEE, 2021.

[36] Dominik Kreuzberger, Niklas Kühl, and Sebastian Hirschl. Machine learning operations (mlops): Overview, definition, and architecture. *IEEE Access*, 11:31866–31879, 2023.

[37] Aviral Kumar, Aurick Zhou, George Tucker, and Sergey Levine. Conservative q-learning for offline reinforcement learning. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020.

[38] Valliappa Lakshmanan, Sara Robinson, and Michael Munn. *Machine Learning Design Patterns*. O'Reilly Media, Inc., 2020.

[39] Nevena Lazic, Craig Boutilier, Tyler Lu, Eehern Wong, Binz Roy, M. K. Ryu, and Greg Imwalle. Data center cooling using model-predictive control. In Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, pages 3818–3827, 2018.

[40] Jörg Lenhard, Mohammad Mahdi Hassan, Martin Blom, and Sebastian Herold. Are code smell detection tools suitable for detecting architecture degradation? In *Proceedings of the 11th European Conference on Software Architecture: Companion Proceedings*, pages 138–144, 2017.

[41] Eric Liang, Richard Liaw, Robert Nishihara, Philipp Moritz, Roy Fox, Ken Goldberg, Joseph Gonzalez, Michael I. Jordan, and Ion Stoica. Rllib: Abstractions for distributed reinforcement learning. In Jennifer G. Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, volume 80 of *Proceedings of Machine Learning Research*, pages 3059–3068. PMLR, 2018.

[42] Penghao Liang, Bo Song, Xiaoan Zhan, Zhou Chen, and Jiaqiang Yuan. Automating the training and deployment of models in mlops by integrating systems with machine learning. *arXiv preprint arXiv:2405.09819*, 2024.

[43] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. In Yoshua Bengio and Yann LeCun, editors, *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016.

[44] Jie Lu, Anjin Liu, Fan Dong, Feng Gu, João Gama, and Guangquan Zhang. Learning under concept drift: A review. *IEEE Trans. Knowl. Data Eng.*, 31(12):2346–2363, 2019.

[45] Sasu Mäkinen, Henrik Skogström, Eero Laaksonen, and Tommi Mikkonen. Who needs mlops: What data scientists seek to accomplish and how can mlops help? In *1st IEEE/ACM Workshop on AI Engineering - Software Engineering for AI, WAIN@ICSE 2021, Madrid, Spain, May 30-31, 2021*, pages 109–112. IEEE, 2021.

[46] Iñigo Martinez, Elisabeth Viles, and Igor G. Olaizola. Data science methodologies: Current challenges and future approaches. *Big Data Res.*, 24:100183, 2021.

[47] Amir M. Mir, Evaldas Latoskinas, Sebastian Proksch, and Georgios Gousios. Type4py: Practical deep similarity learning-based type inference for python. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*, pages 2241–2252. ACM, 2022.

[48] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. *CoRR*, abs/1602.01783, 2016.

[49] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013.

[50] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, et al. Ray: A distributed framework for emerging {AI} applications. In *13th USENIX symposium on operating systems design and implementation (OSDI 18)*, pages 561–577, 2018.

[51] Nuthan Munaiah, Steven Kroh, Craig Cabrey, and Meiyappan Nagappan. Curating github for engineered software projects. *Empirical Software Engineering*, 22:3219–3253, 2017.

[52] Nadia Nahar, Haoran Zhang, Grace A. Lewis, Shurui Zhou, and Christian Kästner. A meta-summary of challenges in building products with ML components - collecting experiences from 4758+ practitioners. In *2nd IEEE/ACM International Conference on AI Engineering - Software Engineering for AI, CAIN 2023, Melbourne, Australia, May 15-16, 2023*, pages 171–183. IEEE, 2023.

[53] Arun Nair, Praveen Srinivasan, Sam Blackwell, Cagdas Alcicek, Rory Fearon, Alessandro De Maria, Vedavyas Panneershelvam, Mustafa Suleyman, Charles Beattie, Stig Petersen, Shane Legg, Volodymyr Mnih, Koray Kavukcuoglu, and David Silver. Massively parallel methods for deep reinforcement learning. *CoRR*, abs/1507.04296, 2015.

[54] Evangelos Ntentos, Stephen J Warnett, and Uwe Zdun. Supporting architectural decision making on training strategies in reinforcement learning architectures. 2024.

[55] Afshin Oroojlooy and Davood Hajinezhad. A review of cooperative multi-agent deep reinforcement learning. *Appl. Intell.*, 53(11):13677–13722, 2023.

[56] Michael Pradel, Georgios Gousios, Jason Liu, and Satish Chandra. Typewriter: neural type prediction with search-based validation. In Prem Devanbu, Myra B.

Cohen, and Thomas Zimmermann, editors, *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, pages 209–220. ACM, 2020.

[57] Antonin Raffin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus, and Noah Dormann. Stable-baselines3: Reliable reinforcement learning implementations. *J. Mach. Learn. Res.*, 22:268:1–268:8, 2021.

[58] Gregorio Robles, Raula Gaikovina Kula, Chaiyong Ragkhitwetsagul, Tattiya Sakulniwat, Kenichi Matsumoto, and Jesus M Gonzalez-Barahona. pycefr: Python competency level through code analysis. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*, pages 173–177, 2022.

[59] Mohammad Reza Samsami and Hossein Alimadad. Distributed deep reinforcement learning: An overview. *CoRR*, abs/2011.11012, 2020.

[60] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017.

[61] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT press, 2018.

[62] Georgios Symeonidis, Evangelos Nerantzis, Apostolos Kazakis, and George A. Papakostas. Mlops - definitions, tools and challenges. In *12th IEEE Annual Computing and Communication Workshop and Conference, CCWC 2022, Las Vegas, NV, USA, January 26-29, 2022*, pages 453–460. IEEE, 2022.

[63] Justin K. Terry, Benjamin Black, Nathaniel Grammel, Mario Jayakumar, Ananth Hari, Ryan Sullivan, Luis S. Santos, Clemens Dieffendahl, Caroline Horsch, Rodrigo Perez-Vicente, Niall L. Williams, Yashas Lokesh, and Praveen Ravi. Pettingzoo: Gym for multi-agent reinforcement learning. In Marc'Aurelio Ranzato, Alina Beygelzimer, Yann N. Dauphin, Percy Liang, and Jennifer Wortman Vaughan, editors, *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*, pages 15032–15043, 2021.

[64] Carmine Vassallo, Sebastian Proksch, Anna Jancso, Harald C Gall, and Massimiliano Di Penta. Configuration smells in continuous delivery pipelines: a linter and a six-month study on gitlab. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 327–337, 2020.

[65] Qing Wang, Jiechao Xiong, Lei Han, Peng Sun, Han Liu, and Tong Zhang. Exponentially weighted imitation learning for batched historical data. In Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi,

and Roman Garnett, editors, *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, pages 6291–6300, 2018.

[66] Stephen John Warnett and Uwe Zdun. Architectural design decisions for machine learning deployment. In *2022 IEEE 19th International Conference on Software Architecture (ICSA)*, pages 90–100. IEEE, 2022.

[67] Bo Yang and Min Liu. Keeping in touch with collaborative uavs: A deep reinforcement learning approach. In Jérôme Lang, editor, *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden*, pages 562–568. ijcai.org, 2018.

[68] Kaiqing Zhang, Zhuoran Yang, and Tamer Basar. Multi-agent reinforcement learning: A selective overview of theories and algorithms. *CoRR*, abs/1911.10635, 2019.

[69] Guanjie Zheng, Fuzheng Zhang, Zihan Zheng, Yang Xiang, Nicholas Jing Yuan, Xing Xie, and Zhenhui Li. DRN: A deep reinforcement learning framework for news recommendation. In Pierre-Antoine Champin, Fabien Gandon, Mounia Lalmas, and Panagiotis G. Ipeirotis, editors, *Proceedings of the 2018 World Wide Web Conference on World Wide Web, WWW 2018, Lyon, France, April 23-27, 2018*, pages 167–176. ACM, 2018.

A. Appendix

ID	Title	Url
S1	What is Reinforcement Learning?	https://www.mathworks.com/help/reinforcement-learning/ug/what-is-reinforcement-learning.html
S2	Introduction: Reinforcement Learning with OpenAI Gym	https://towardsdatascience.com/reinforcement-learning-with-openai-d445c2c687d2
S3	Getting Started With OpenAI Gym: The Basic Building Blocks	https://blog.paperspace.com/getting-started-with-openai-gym/
S4	Hands-On Introduction to Reinforcement Learning in Python	https://towardsdatascience.com/hands-on-introduction-to-reinforcement-learning-in-python-da07f7aaca88
S5	An Introduction to Reinforcement Learning with OpenAI Gym, RLlib, and Google Colab	https://www.anyscale.com/blog/an-introduction-to-reinforcement-learning-with-openai-gym-rllib-and-google#what-is-reinforcement-learning
S6	Intro to RLlib: Example Environments	https://medium.com/distributed-computing-with-ray/intro-to-rllib-example-environments-3a113f532c70
S7	An actually runnable tutorial for getting started with gymnasium and reinforcement learning	https://www.sliceofexperiments.com/p/an-actually-runnable-march-2023-tutorial
S8	Chapter 4. Reinforcement Learning with Ray RLlib	https://www.oreilly.com/library/view/learning-ray/9781098117214/ch04.html
S9	A Recipe for Training Neural Networks	https://karpathy.github.io/2019/04/25/recipe/
S10	Lessons Learned Reproducing a Deep Reinforcement Learning Paper	http://amid.fish/reproducing-deep-rl
S11	Deep Reinforcement Learning: Pong from Pixels	https://karpathy.github.io/2016/05/31/rl/
S12	Early Stopping in Machine Learning	https://aoishidas28.medium.com/early-stopping-in-machine-learning-16b71542f29b
S13	Deep Reinforcement Learning Doesn't Work Yet	https://www.alexirpan.com/2018/02/14/rl-hard.html
S14	Multi-Input Gymnasium Envs and Stable-Baselines3 Agents	https://marc-velay.medium.com/custom-gymnasium-envs-and-multi-input-stable-baselines3-agents-986d84a27e7d
S15	Reinforcement Learning Tips and Tricks	https://stable-baselines.readthedocs.io/en/master/guide/rl_tips.html
S16	RL — Tips on Reinforcement Learning	https://jonathan-hui.medium.com/rl-tips-on-reinforcement-learning-fbd121111775
S17	Reinforcement Learning Implementation Tips and Tricks	https://agents.inf.ed.ac.uk/blog/reinforcement-learning-implementation-tricks/
S18	Best practices for Reinforcement Learning	https://towardsdatascience.com/best-practices-for-reinforcement-learning-1cf8c2d77b66
S19	MLOps Basics	https://github.com/graviraja/MLOps-Basics
S20	Implementing MLOps with GitHub Actions	https://medium.com/@craftworkai/implementing-mlops-with-github-actions-acd007e47074
S21	Set up MLOps with GitHub	https://learn.microsoft.com/en-us/azure/machine-learning/how-to-setup-mlops-github-azure-ml
S22	MLOps with GitHub actions	https://lo-victoria.com/series/mlops
S23	CI/CD for MLOps on GitLab	https://medium.com/marvelous-mlops/ci-pipeline-for-mlops-on-gitlab-part-1-19ac79737fab

Table A.1.: Grey literature used for the investigated patterns and practices. Note that all the provided URLs were accessed in September 2024.

Algorithm	**RLlib**	**SB3**
A2C [48]		✔
APPO [1]	✔	
BC [65]	✘	
CQL [37]	✘	
DDPG [43]		✔
DQN [49]	✔	✔
DreamerV3 [27]	✔	
HER [6]		✔
Impala [19]	✔	
MARWIL [19]	✘	
PPO [60]	✔	✔
SAC [25]	✔	✔
TD3 [23]		✔

Table A.2.: Distributed nature of the analysed algorithms for RLlib and stable-baselines3.
✔= distributed, ✘ = non distributed, missing = not implemented

Environment	Fully Cooperative	Fully Competitive	Mixed Cooperative-Competitive
basketball_pong		✔	
boxing		✔	
combat_jet		✔	
combat_tank		✔	
double_dunk		✔	
entombed_competitive		✔	
entombed_cooperative	✔		
flag_capture		✔	
foozpong			✔
ice_hockey		✔	
joust		✔	
mario_bros			✔
maze_craze		✔	
othello		✔	
pong		✔	
quadrapong			✔
space_invaders			✔
space_war		✔	
surround		✔	
tennis		✔	
video_checkers		✔	
volleyball_pong			✔
warlords		✔	
wizard_of_wor		✔	
cooperative_pong	✔		
knights_archers_zombies	✔		
pistonball	✔		
chess		✔	
connect_four		✔	
gin_rummy		✔	
go		✔	
hanabi	✔		
leduc_holdem		✔	
rps		✔	
texas_holdem_no_limit		✔	
texas_holdem		✔	
tictactoe		✔	
simple_adversary			✔
simple_crypto			✔
simple_push		✔	
simple_reference	✔		
simple_speaker_listener	✔		
simple_spread	✔		
simple_tag			✔
simple_world_comm			✔
multiwalker	✔		
pursuit	✔		
waterworld			✔

Table A.3.: Classification of the MARL environments present in `pettingzoo`. Note that these characteristics are the default ones since some environments allow a user-defined number of agents, which may change the environment type.

RL Category	Element	Technology based	Heuristic based
Checkpoint	`CheckpointCallback` object instantiation	✔	
Checkpoint	Assignment on `saving_frequency` variable		✔
Hyperparameter Tuning	`ray.tune.run` function call	✔	
Hyperparameter Tuning	`do_tuning` function definition		✔
Distributed RL	`PPO.load` function call	✔	
Distributed RL	Assignment on `workers` variable		✔
Single-Agent RL	`gym.make` function call	✔	
Single-Agent RL	Assignment on `env` variable		✔
Multi-Agent RL	`pettingzoo.AECEnv` subclass	✔	
Multi-Agent RL	`MARLEnv` object instantiation		✔
RL Pipeline	`learn` method call	✔	
RL Pipeline	`evaluate_agent` function definition		✔
Training Configuration	`numpy.random.seed` function call	✔	
Training Configuration	`gamma` keyword usage		✔

Table A.4.: Examples of defined elements for the detection of RL patterns and practices.