universität wien

MASTERARBEIT | MASTER'S THESIS

Titel | Title

Detection of Code Smells in Reinforcement Learning Applications

verfasst von | submitted by

Ing. Georgiana Teodora Filip

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

Master of Science (MSc)

Wien | Vienna, 2026

Studienkennzahl lt. Studienblatt | Degree programme code as it appears on the student record sheet:	UA 066 935
Studienrichtung lt. Studienblatt | Degree programme as it appears on the student record sheet:	Masterstudium Medieninformatik
Betreut von | Supervisor:	Univ.-Prof. Dr. Uwe Zdun

Acknowledgements

I would first like to thank my supervisor, Univ.-Prof. Dr. Uwe Zdun, for his guidance, feedback, and support throughout this thesis. His advice and experience were extremely valuable during both the research, implementation of the project and writing process.

I would also like to give a special and big thank you to Deepansha Chowdhary, MSc, for her continuous support, patience, and involvement throughout this project. Her feedback, encouragement, and willingness to discuss ideas at every stage of the work had a major impact on the development of this thesis, and I am very grateful for her help and guidance.

Special thanks go to my family, without whose support, from the very beginning until the end of this journey, this work would not have been possible.

Abstract

Reinforcement learning is one of the three basic machine learning paradigms, alongside supervised learning and unsupervised learning. It is an area concerned with the optimal decision-making of an agent and how an intelligent agent should take action in a dynamic environment to maximize a reward signal. The core problem in reinforcement learning focuses on how an intelligent agent can make a good sequence of decisions, evaluate what constitutes a good decision, and learn to reach the optimal solution and maximize the long-term reward. Such systems and projects are often developed by data analysts who might not always have a software engineering background. Therefore, it is important to analyze possible bottlenecks, bugs, or code issues, referred to as code smells from a software engineering perspective. Identifying and improving code smells is crucial for maintaining a well-structured, scalable, secure, and robust reinforcement learning system. The scope of this master thesis is to find the most common code smells, categorize them and provide a practical solution on how to detect them, in order to improve the code quality of the reinforcement learning applications. The work of this thesis is structured in two main parts. The first part, serves as foundation work for the second part, and it focuses on defining and labeling reinforcement learning specific code smells on a limited set of project, manually analyzing source code and compiling a solid list of reinforcement learning specific code smells. We defined six different categories of reinforcement specific code smells, based on the areas they impacts most in the reinforcement learning process: HYPERPARAMETER, TRAINING, EVALUATION, CODESTYLE, AGENT, and ENVIRONMENT. The second part of the thesis consists of building a tool to automate the process of labeling reinforcement learning specific code smells. The tool was applied to 51 open-source repositories spanning different application domains, resulting in 4,872 positive detections. HYPERPARAMETER smells are the most prevalent category (29.95%), followed by EVALUATION (24.98%), TRAINING (22.66%), and CODESTYLE (19.87%). The distribution of code smells categories is consistent across domains, suggesting that these quality issues occur regularly in reinforcement learning projects, regardless of application area. This work represents a stepping stone towards systematic quality assessment of reinforcement learning codebases, providing automated detection of implementation anti-patterns that directly affect reproducibility and experimental validity, can lead to bugs and suboptimal performance of the systems.

Kurzfassung

Reinforcement Learning ist neben überwachtem und unüberwachtem Lernen eines der drei grundlegenden Paradigmen des maschinellen Lernens. Es befasst sich mit der optimalen Entscheidungsfindung eines Agenten und untersucht, wie ein intelligenter Agent in einer dynamischen Umgebung handeln sollte, um ein Belohnungssignal zu maximieren. Das Kernproblem des Reinforcement Learning besteht darin, dass ein Agent lernen muss, eine gute Abfolge von Entscheidungen zu treffen, zu bewerten, was eine gute Entscheidung ausmacht, und eine Strategie zu entwickeln, die langfristig den erwarteten Gesamtertrag maximiert. Solche Systeme und Projekte werden häufig von Datenanalystinnen und Datenanalysten oder Forschenden entwickelt, die nicht immer über einen Hintergrund in der Softwareentwicklung verfügen. Daher ist es wichtig, mögliche Engpässe, Fehler oder Code-Probleme zu analysieren, die aus Sicht der Softwareentwicklung als 'Code Smells' bezeichnet werden. Das Erkennen und Beheben solcher Code Smells ist entscheidend für die Entwicklung gut strukturierter, skalierbarer, sicherer und robuster Reinforcement-Learning-Systeme.

Ziel dieser Masterarbeit ist es, häufig auftretende Code Smells in Reinforcement-Learning-Anwendungen zu identifizieren, zu kategorisieren und eine praktische Lösung für ihre automatisierte Erkennung bereitzustellen. Dadurch soll die Codequalität von Reinforcement-Learning-Anwendungen verbessert werden. Die Arbeit gliedert sich in zwei Hauptteile. Der erste Teil dient als Grundlage für den zweiten Teil und konzentriert sich darauf, Reinforcement-Learning-spezifische Code Smells anhand einer begrenzten Anzahl von Projekten zu definieren und zu kennzeichnen. Dazu wurde der Quellcode ausgewählter Projekte manuell analysiert, um eine fundierte Liste wiederkehrender Reinforcement-Learning-spezifischer Code Smells zu erstellen. Auf dieser Grundlage wurden sechs Kategorien von Code Smells definiert, die sich an den Bereichen orientieren, die sie im Reinforcement-Learning-Prozess am stärksten beeinflussen: HYPERPARAMETER, TRAINING, EVALUATION, CODESTYLE, AGENT und ENVIRONMENT.

Der zweite Teil der Arbeit umfasst die Entwicklung eines Tools zur Automatisierung der Erkennung von Reinforcement-Learning-spezifischen Code Smells. Das Tool wurde auf 51 Open-Source-Repositories aus verschiedenen Anwendungsbereichen angewendet. Insgesamt wurden dabei 4.872 Code-Smell-Instanzen identifiziert. Die Kategorie 'HYPERPARAMETER' war mit 29,9 % am stärksten vertreten, gefolgt von 'EVALUATION' mit 25,0 %, 'TRAINING' mit 22,7 % und 'CODESTYLE' mit 19,9 %. Die Verteilung der Code-Smell-Kategorien zeigte sich über die untersuchten Domänen hinweg weitgehend konsistent. Dies deutet darauf hin, dass entsprechende Qualitätsprobleme in Reinforcement-Learning-Projekten regelmäßig auftreten und nicht auf einzelne Anwendungsbereiche beschränkt sind.

Die Arbeit leistet damit einen Beitrag zur systematischen Qualitätsbewertung von

Reinforcement-Learning-Codebasen. Sie zeigt, dass sich bestimmte Implementierungs-Anti-Patterns automatisiert erkennen lassen, insbesondere solche, die die Reproduzierbarkeit, die experimentelle Validität sowie die Zuverlässigkeit und Leistungsfähigkeit von Reinforcement-Learning-Systemen beeinträchtigen können.

Contents

1. Introduction

In this thesis, the main objective is to explore the types of code smells that can occur in reinforcement learning applications, provide a categorization, and study the correlations between the code smells and different characteristics of the projects. It is designed in two phases: the first phase consists of a manual analysis of a subset of open-source Reinforcement learning (RL) GitHub projects, alongside defining potential code smells that are specific to the RL pipeline, and finally, building a prototype Command-Line Interface (CLI) tool to automate the code smell detection. In the second phase, we defined categories for the code smells, extended the CLI tool to detect additional code smells, added new features and a proper user-interface, used the tool to analyze 51 open source RL GitHub projects, and examined the results.

1.1. Objective

This study aims to identify and understand code smells in reinforcement learning projects, with a focus on the most common occurrences and their impact on software quality. Code smells are indicators of potential structural or design issues that can make software harder to maintain, understand, scale, and extend over time. While the concept of code smells has been studied extensively in general-purpose software and in deep learning applications [14], reinforcement learning projects present unique challenges that make these studies insufficient. RL pipelines often involve complex training loops, iterative experimentation, dynamic environments, and extensive parameter tuning, which can lead to structural patterns and smells that are uncommon in other domains. By investigating which code smells occur most frequently, how they manifest in different stages of the RL workflow, and how they relate to maintainability and robustness, this study seeks to provide practical guidance for developers and researchers working with RL systems and to bridge a clear gap in the empirical literature on RL-specific software quality issues.

- Identify code smells in RL systems: analyze a set of projects with a focus on identifying RL specific code smells.
- Spot trends and patterns: identify recurring code smells, particularly those unique to RL related programming.
- Detect the prevalence of code smells across the different stages of a reinforcement learning pipeline

By exploring this area, we hope to shed light on the challenges of writing clean, maintainable RL code and provide useful recommendations to the community.

1.2. Motivation

Reinforcement learning is becoming a game-changer in fields such as robotics [18], gaming [13], autonomous driving systems [17], stock trading [19] and many more. However, with this growth comes a big challenge: high standards for the quality of the code for RL projects. Code quality [31] is a highly important topic for developers as well as organizations, since good quality software lowers the overall technical debt, can lead to a good developer experience, increases software development velocity, reduces the risks of operational bugs and many more [S1]. Many RL codebases, especially in research settings, are often rushed, leading to low code maintainability, insufficient error handling, poor design decisions, low configurability, etc. Over time, these issues can make the code harder to work with, leading to bugs and wasted time and resources. [16]

Furthermore, reinforcement learning projects exhibit characteristics that distinguish them from conventional software systems [23]. They typically involve complex training loops, experimentation-heavy workflows, dynamic environments, extensive parameter tuning, and non-trivial initialization procedures. Maintaining clean code is not easy [40], as code is frequently adapted, extended, or reconfigured during experimentation, which increases structural complexity over time. These properties can amplify traditional code smells or even give rise to RL-specific ones that are less common in other domains. Despite the growing importance of reinforcement learning in both research and industry, limited attention has been given to systematically studying the maintainability challenges that arise from these unique development patterns.

During our preliminary analysis, we observed a clear gap in the literature regarding code smells that are specific to reinforcement learning systems. While general-purpose and Machine learning (ML) code smells related have been studied and there are plenty of tools to identify them, they do not fully capture the structural and experimental nature of RL codebases. This gap motivates our investigation of the types and impact of code smells in RL projects, with the broader goal of supporting researchers and developers in improving software quality and long-term maintainability. In addition to this gap, it remains largely unexplored which code smells occur most frequently in RL projects, whether certain smells dominate others, and at which stages of the RL pipeline they tend to emerge (e.g., environment setup, training loops, evaluation, or hyperparameter configuration). These open questions motivated a systematic investigation into both the types and distribution of code smells in reinforcement learning systems.

By identifying recurring code smells in RL projects and analyzing the conditions under which they occur, this thesis aims to provide practical insights for writing cleaner and more reliable RL software. Ultimately, improving code quality in RL systems allows developers to focus on advancing algorithms and experimentation rather than struggling with code quality issues in the code base.

2. Fundamentals

2.1. Reinforcement Learning

RL is a branch of machine learning that is concerned with an autonomous agent that learns to make decisions by interacting with their environment.

Reinforcement learning [37] provides a strong framework for training intelligent agents, but also introduces unique challenges with respect to code complexity, performance, and structure. RL algorithms can become highly sophisticated and complex, therefore, machine learning engineers and developers must carefully manage and structure the code to avoid potential issues and architectural designs problems that can hinder progress, make the system difficult to maintain, or conceal deeper performance, security, and logic problems.

2.1.1. Concepts

One of the main questions RL tries to answer is how can an intelligent agent make a good sequence of decisions?. Therefore, as a follow-up, we naturally ask what is a good decision?. The agent does so by learning, it does not know at the beginning what is a good or a bad decision. In RL, there are a few core concepts [32] that we need to introduce before going into a deeper analysis, those being the following:

- **Agent**: the decision-maker component
- **Environment**: what the agent interacts with
- **State (S)**: a current description of the environment [32]
- **Actions (A)**: an action or decision the agent makes in the environment
- **Reward (R)**: the feedback of the environment to the agent, following a decision, it incorporates how good or bad a decision the agent took was based on the scope. The reward is a scalar feedback signal.
- **Policy (π)**: the strategy used by the agent to decide an action, based on the current state, or as [32] describes it in their paper: "The agent's policy (the policy function) is the mapping from possible states to possible actions."
- **Value function (V)**: the function that estimates the expected cumulative reward from a given state. Sutton et al. also put it in this way: "The value function is the agent's current mapping from the set of possible states (or state-action pairs) to its

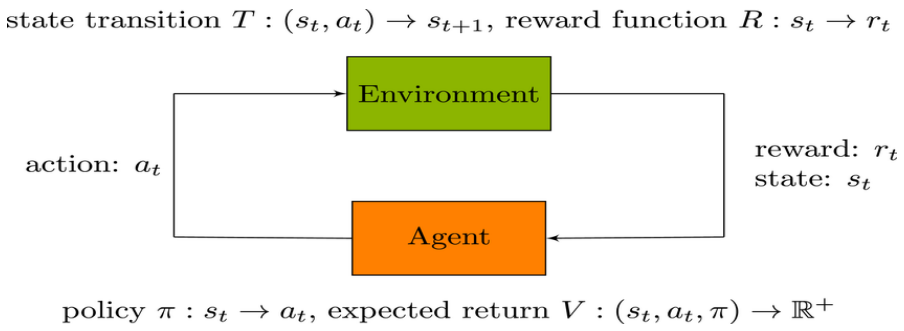

Figure 2.1.: Simple representation of RL concepts [10]

> estimates of the net long term reward to be expected after visiting a state (or a state-action pair) and continuing to act according to its current policy thereafter." [32]

In simple terms, RL is a strong paradigm of learning, where the agent learns the world it moves in, by performing and taking certain decisions, and is rewarded immediately. Based on these rewards, whether they are more or less substantial, the agent learns by itself which actions are better to do in order to obtain the best results [12]. Ghasemi et al. illustrate these concepts in a simple example about a maze [12]: they illustrate the state as the current position within the maze, where the agent is able to move north, south, east or west. They correlate the policy to the function that dictates how the agent should move based on the current state. The reward function gives, for example, a positive reward when the agent reaches the end of the maze and a negative reward when the agent hits a wall [12].

In RL, the end goal can be described by the maximization of expected cumulative reward. History is the sequence of states, often referred to also as observations, actions and rewards, therefore what happens next depends on the history: what action the agent selects, and the environment selects the states and the reward.

2.1.2. Markov Decision Process

Behind RL lies the Markov Decision Process. In RL, the environment has to be modeled as Markov Decision Process (MDP). MDP is a mathematical framework used for modeling an environment [1] and guides how decisions are evaluated over time, defining how the agent interacts with an environment through states, actions, and rewards, for learning the best behavior, in order to reach the optimal solution. ([S2]).

$$\text{MDP} = (S, A, P, R, \gamma).$$

1. States The state space S represents all possible situations in which the agent can exist. It is defined as a finite set of states:

$$S = \{s_1, s_2, \ldots, s_N\}, \quad |S| = N.$$

Each state provides the information necessary for the agent to decide which action to take next.

2. Actions The action space A is the finite set of actions available to the agent:

$$A = \{a_1, a_2, \ldots, a_K\}, \quad |A| = K.$$

Here, $K = |A|$ denotes the total number of available actions. For a given state $s \in S$, the set of admissible actions is denoted by $A(s)$, where

$$A(s) \subseteq A.$$

This means that not all actions must be available in every state [S2].

3. Transition Function The transition function P defines the probability of moving from one state to another. More formally, it describes the probability of transitioning to state s' at time step $t + 1$, given that the current state at time t is s:

$$P_{s,s'} = \Pr(S_{t+1} = s' \mid S_t = s).$$

In the transition matrix P, the sum of each row equals 1, since the probabilities of all possible successor states must add up to one [S3].

4. Reward Function The reward function provides numerical feedback that guides the agent toward the optimal objective [S2]. It evaluates how well the agent performs at each step [S3]. The reward function assigns positive, negative, or zero values to outcomes, thereby encouraging desirable actions and discouraging undesirable ones [S2]. Formally, the reward function is defined as:

$$R : S \times A \times S \to \mathbb{R}.$$

5. Discount Factor The discount factor γ determines the importance of future rewards relative to immediate rewards and satisfies

$$\gamma \in [0, 1].$$

It is introduced because:

- real-world environments are often uncertain or imperfectly modeled,
- future rewards should not accumulate indefinitely,
- uncertainty about the future should be reflected in the decision process [S3].

2.1.3. Challenges

Reinforcement learning faces some critical challenges that need to be addressed and implemented. One of those is the exploration vs. exploitation trade-off, which describes the balance between the agent trying new actions in order to determine their effects and gather additional information, and exploitation, that involves the agent to use its current knowledge to make decisions that maximize immediate reward [12]. In RL, there are certain steps that need to be taken and implemented in order to obtain the best results out of the agent and the systems. In this master thesis we also explore these steps and analyzed them from a code smell perspective.

2.2. Code Smells

Code smells are a metaphorical [39] term used to describe various types of concerns and problems that arise in code. As Martin Fowler describes them, "code smells are symptoms of poor design or implementation choices " [8]. The term is widely used to identify distinct issues in a system, such as code style violations, type mismatches, large classes, and long methods, that can lead to poor maintainability [38], limited scalability, bugs, and even security vulnerabilities. The impact of code smells on software systems has been extensively studied [24] and remains an active area of research in software engineering. Code smells are violations of coding standards that make code harder to read and maintain. They can also hide potential vulnerabilities, leading to a less secure, less scalable, more inefficient, and more prone to failure system. Common code smells [9], such as long methods, large classes, coding standard violations, hard-coded values, unused libraries or modules, nested loops, and duplicated code, are easily detected using static code analysis tools, which have been developed for a wide range of programming languages. However, these general categories of code smells are not well-suited for this study, as they do not capture concerns specific to RL concepts. In the RL context, code smells can lead to suboptimal training, poor system performance, and unreliable results. Representative examples include the absence of hyperparameter tuning (e.g., the learning rate, the number of training episodes), improper episode termination (which can introduce inconsistencies across training runs), and a flawed reward calculation (which prevents the agent from learning effectively). Accordingly, this study focuses primarily on RL pipeline code smells rather than general Python code smells detectable by standard static analysis tools.

Early detection of code smells and faulty design decisions is therefore critical in RL applications. Complex state spaces and large-scale environments make this challenge even more difficult, as the learning process must carefully balance exploration and exploitation. Addressing code smells in RL codebases is essential to ensure that learning algorithms remain efficient, scalable, and open to modifications or extensions over time. For instance, duplicated logic or excessively long functions constrain comprehension and maintenance of RL code, while redundant or inefficient algorithms can cause suboptimal agent performance and unnecessarily prolonged training.

3. Related Work

Code smells are indicators of potential issues in code that, over time, make it harder to maintain, more prone to bugs, and more difficult to read and manage. Dealing with smelly code often leads to wasted time, resources, and significant frustration, making the pursuit of clean and maintainable code essential. However, existing work in ML and RL tends to focus on detecting general code smells, such as common Python issues, long methods, low modularity, or duplicated code, without considering the unique challenges posed by RL systems. This work aims to bridge that gap by identifying and analyzing RL specific code smells, that may conceal critical issues in RL systems.

3.1. Code Smell Detection via Deep Learning

An important consideration in this area is the inherent difficulty of detecting code smells reliably. Liu et al. [21] explored the challenges of automated code smell detection in Java projects, proposing a deep learning approach to address these limitations. Their results demonstrated a notable improvement in detection accuracy of up to 48.18%, suggesting that deep learning techniques can contribute meaningfully to more efficient software maintenance and better adherence to coding standards. The authors further noted that this methodology could be extended to a broader range of smells and integrated into modern development tools and pipelines.

A related study by van Emden et al. [34] investigated how automated detection and visualization of code smells can improve software quality assurance. The authors developed a prototype tool called jCOSMO to support code smell detection in Java programs through automatic static code analysis, targeting smells such as long methods and large classes. Beyond detection, the tool provides a graph-based visualization that clusters specific smells, such as typecasting and improper use of switch statements, facilitating easier identification and review.

Existing research primarily focuses on static code smells in ML applications. Van Oort et al. [35] analyzed code smells in Python-based ML projects by building an analysis tool that clones each project's latest Git repository, sets up a virtual environment, and installs the listed dependencies. This setup enables static analysis to validate imports and library usage, which is particularly important in ML projects that rely heavily on external libraries [3]. By applying the static analysis tool Pylint [S31] to 74 projects, the study identified the most prevalent code smells both overall and per category, with code duplication emerging as a particularly frequent concern, requiring further investigation. The authors also observed that PEP8 [S23] naming conventions may not always align with ML code, as they can conflict with standard mathematical notation.

The study further identified critical issues with dependency specifications that threaten reproducibility and maintainability, and noted that Pylint's high false-positive rate for import statements limits its practical effectiveness in ML projects [35]. These challenges also hinder the adoption of continuous integration practices and underscore the need for improved dependency management. The smells most commonly detected included "unused-wildcard-import", "bad-indentation", "invalid-name", "line-too-long", "missing-function-docstring", "no-member", "duplicated-code", "trailing-whitespace", "redefined-outer-name", and "missing-module-docstring" [35]. While this work provides extensive coverage of static code smells in general ML projects, it does not address design-related code smells specific to RL systems, a gap that this study aims to fill.

3.2. Code Smells in Reinforcement Learning Projects

Another relevant study was conducted by Cardozo et al. [2]. In their study, the authors analyzed 24 Python-based RL projects, including 20 GitHub repositories and 4 professional projects based on the ACME framework. Their objective was to identify common code smells and examine their impact on the maintainability, scalability, security, and robustness of RL systems. To support this analysis, they used eight software metrics related to code organization, abstraction, and expression. Overall, the study identified 1,090 code smells, indicating that maintainability-related issues are widespread in RL projects.

The most frequent code smells included:

- Long methods: functions that handle too many operations, making them harder to read and maintain.
- Large classes: classes that have too many responsibilities, violating modularity principles.
- Multiply-nested containers: deeply nested data structures that are difficult to work with and prone to errors.
- Long parameter lists: functions with an excessive number of parameters, leading to fragile, error-prone code and reduced usability.

Interestingly, GitHub repository projects had about 3.15 code smells per file, much higher than the 0.8 smells per file in the professionally developed ACME projects. However, both types of projects shared the same recurring smells, suggesting that some of these issues are related to the complexity of RL itself. Among these are the challenges of representing environments, states, and transitions.

The study emphasized the need for better tools and practices in RL development. Specifically, the authors called for dedicated static analysis tools with RL specific metrics to identify and address these smells more effectively. They also proposed introducing new programming abstractions or even domain-specific languages tailored to RL's unique challenges. Such tools and frameworks could make RL projects easier to maintain, extend, and debug, paving the way for higher-quality software in this rapidly growing field.

Ntentos et al., in their study [23] introduced a structured Architectural Design Decision model that identifies six key decisions: model architecture, training strategies, checkpoints, transfer learning, distribution strategies, and hyperparameter tuning. These decisions serve as a systematic foundation for detecting RL specific code smells by bridging theoretical insights with practical applications and emphasizing scalability, efficiency, and adaptability. The study also highlights critical patterns such as multi-agent systems and hierarchical models.

The key highlights of the research include discussions on model architectures (like monolithic and multi-agent systems), training strategies (such as centralized vs. decentralized approaches), and the role of checkpoints, transfer learning, distributed strategies, and hyperparameter tuning. The work emphasizes bridging the gap between theoretical insights and practical implementation in RL. It also provides structured guidance for RL developers, aiming to enhance efficiency, adaptability, and scalability while managing the complexity of RL-based systems.

3.3. Code Smells for Machine Learning Applications

In the paper "Code Smells for Machine Learning Applications" [41], Zhang et al. argue about the importance of code quality in ML projects, which is also a fundamental basis for this thesis: with the rise of ML applications, which are often integrated into complex systems, there is a need to ensure proper code quality, to avoid long-term issues [41]. Their goal is to eliminate software quality pitfalls that can lead to substantial issues over time, and therefore eliminating those during the development stage is considered crucial, which we also agree with.

In order to assess their impact, Zhang et al. formulate the following research question: *"What are the recurrent code issues that may arise from the peculiarities of machine learning applications?"* [41]. To answer this question, they state two main contributions of their paper:

- a catalog of machine-learning-specific code smells
- a large dataset of 1750 papers, 2170 grey literature entries, 87 GitHub commits and 491 Stack Overflow posts for empirical studies. [41]

Their methodology involved gathering information from four different sources: academic papers, GitHub bug-fix commits, blogs, and Stack Overflow posts, looking for major ML libraries. After this, they performed a cross-check for every candidate that was flagged as a code smell, then the authors independently reviewed and debated the full list, constructing a catalog of 22 ML specific code smells. The study also categorized each smell by its effect, such as whether it causes errors, harms performance, breaks reproducibility, wastes memory or reduces readability.

The authors grouped the ML specific code smells across four different pipeline stages:

- Data cleaning (9 smells): issues like unnecessary iteration (using loops instead of vectorized Pandas/NumPy operations), Chain Indexing (e.g., `df["col1"]["col2"]` in Pandas, which is slow and error prone), NaN equivalence and others. [41]

- Feature engineering (1 smell): no scaling before scaling-sensitive operation. [41]
- Model training (9 smells): including hyperparameter not explicitly set, randomness uncontrolled, memory not freed. [41]
- Model evaluation (2 smells): data leakage (training data contaminating validation) and threshold-dependent validation. [41]

Of the 22 smells, 16 are generic (library-agnostic) and 6 are API-specific (tied to TensorFlow, PyTorch, or Pandas). [41]. While their research provides valuable insight into software quality for ML applications, it does not specifically address the gap in RL, which this thesis aims to address.

3.4. ML-Specific Code Smells

Another significant study in the field of ML and code quality is the work by Recupito et al., titled "When code smells meet ML: on the lifecycle of ML-specific code smells in ML-enabled systems" [27]. They build on top of the work done by Zhang et al. [41] and implemented CodeSmile, which is a ML specific code smell detection tool.

In their work, Recupito et al. [27] investigated the lifecycle of ML specific code smells in ML systems through a large-scale study in which they analyzed over 400,000 commits from 337 ML projects. The projects were selected from the NICHE dataset [36], a curated collection of 572 popular and active ML Python projects, requiring each project to have over 100 GitHub stars, at least 100 commits, and recent maintenance activity, with activity no older than May 2020. Due to the high variability in project size, the authors divided projects into three size groups (small, medium, and large) based on lines of code percentiles and they applied sampling within each group at a 95% confidence level and 5% margin of error, resulting in a final sample of 337 projects totaling 401,658 commits.

To enable this investigation, the authors developed CodeSmile, a static analysis tool built on top of the catalog of ML specific code smells defined by Zhang et al. [41]. CodeSmile uses Abstract Syntax Tree (AST) analysis to detect 12 ML code smells. Their tool operates through three main steps: variable extraction and library association, AST traversal of function calls and attribute accesses, and rule-based detection criteria derived directly from the smell definitions of Zhang et al. [41]. The tool was validated through a user study involving ten ML engineers, achieving a precision of 87.4% and a recall of 78.6%, demonstrating that AST static analysis is a viable and effective approach for the automated detection of ML code smells at scale.

In their study, the authors identified 8,542 ML specific code smell instances in the analyzed projects. The most prevalent smell was *Columns and DataType Not Explicitly Set*, accounting for 3,365 occurrences and affecting over 55% of all analyzed projects, followed by *TensorArray Not Used* with 3,085 occurrences. They also examined what predominantly caused the appearance of code smells and they found that ML code smells are most frequently introduced during file modifications rather than file creation, and during new feature development tasks. Smells tend to be introduced in later stages of

project activity, typically more than a year into development and well before upcoming releases, which is an important observation. Regarding smell removal and code cleanup, most smells are resolved during stable maintenance periods via refactoring activities. However, the cumulative trend analysis reveals that introductions consistently outnumber removals across all smell types, meaning that the overall number of active smells grows steadily over time, suggesting that current development practices are insufficient to manage ML specific code smells and improve code quality in ML projects.

This work is significant because it demonstrates that ML code smells are not only theoretical concerns but are genuinely prevalent in real-world ML projects and have measurable consequences for maintainability. However, the study considers ML systems broadly and does not distinguish between different ML paradigms. In particular, reinforcement learning projects present unique structural characteristics, such as the agent-environment interaction loop, reward signal design, and policy and value function definitions, that are unlikely to be captured by a smell catalog designed for general ML pipelines. This gap motivates our present work, which focuses specifically on identifying and characterizing code smells that are specific to RL applications.

To conclude this section, there remains substantial room for further research on code smell detection in the RL context. While the impact of code smells has been widely studied (especially in Java systems) and a few studies have been done in the ML context, there are still many open questions. The focus of this thesis is to detect and analyze the impact of code smells in RL systems.

4. Background

4.1. Reinforcement Learning Pipeline

Reinforcement learning systems typically follow a structured training pipeline composed of several interacting components, from environment setup to training and evaluation. From a high-level perspective, a RL pipeline consists of an environment, an agent, a training loop, reward signals, and evaluation procedures, as already mentioned in the previous chapter. These components work together to enable an agent to learn a policy that maximizes cumulative reward over time, leading to improved behavior and outcomes.

The process begins with the initialization of the environment. The environment represents the problem domain in which the agent operates. The environment provides observations describing the current state and accepts actions from the agent. Based on the chosen action, the environment transitions to a new state and returns a reward signal indicating the quality of the action. Good actions receive higher reward values, while poor actions receive lower reward values. Often, a positive reward value corresponds to a good decision, while a negative reward value indicates a less favorable decision.

The agent is responsible for selecting actions according to a policy function, which may be deterministic or stochastic [5]. During training, the agent interacts with the environment in a loop: it observes the current state of the environment, selects an action from the action space, receives a reward indicating the quality of the action, and updates its internal model based on the experience. This interaction is repeated over multiple episodes until the policy converges or the training budget is exhausted. [5]

A common RL training pipeline includes the following steps:

- Environment initialization
- Agent initialization
- Interaction loop between agent and environment
- Reward processing and learning updates
- Evaluation and logging of performance
- Model checkpoint saving and experiment management

In practice, these stages are implemented through training scripts that orchestrate the learning process. However, RL implementations often introduce engineering challenges due to the stochastic nature of training, the sensitivity to hyperparameters (e.g. learning rate, discount factor), and the complex interaction between components. As a result,

recurring implementation issues can appear in RL codebases, which, ultimately, negatively affect training stability, reproducibility, or maintainability. We refer to these issues as reinforcement learning specific code smells, and our goal is to study them in open-source RL projects, as already mentioned.

4.2. Reinforcement Learning Code Smells

Code smells are indicators of potential design or implementation problems in software systems. Although they do not necessarily represent functional errors, code smells may reduce code maintainability, readability, and long-term system quality.

In the context of reinforcement learning systems, code smells often arise from improper implementation of core elements of the RL pipeline, such as training loops, environment management, reward processing, or experiment configuration. Because RL experiments are computationally expensive and highly sensitive to implementation details, these issues can have significant consequences for training stability, reproducibility, and experimental validity.

Unlike traditional software systems, RL applications introduce additional complexity due to the interaction between machine learning algorithms, simulation environments, and experimental workflows, therefore, certain implementation patterns that may appear harmless in conventional software can lead to serious issues in RL systems. Examples include missing evaluation procedures, absence of checkpoint saving mechanisms, improper environment management, or hard-coded hyperparameters without tuning strategies. These issues do not necessarily cause runtime failures but may lead to unreliable experimental results or inefficient training processes.

In this study, we focus on identifying RL specific code smells that emerge in open-source reinforcement learning repositories. Each code smell represents a recurring implementation pattern that may negatively affect the quality or reliability of RL experiments.

4.3. Research Questions

The objective of this study is to systematically investigate the presence and characteristics of code smells in reinforcement learning (RL) systems. The research questions that we intend to answer through our study are:

- **RQ1**: What types of code smells are most prevalent in reinforcement learning systems?
- **RQ2**: How can we categorize the different code smells found in RL systems?
- **RQ3**: How does the prevalence of code smells correlate with project characteristics such as size, contributor count, age, and other factors?

The first research question aims to identify which RL-specific code smells are most prevalent in open-source projects. In particular, we seeks to determine whether certain

types of code smells occur more frequently than others and to what extent they dominate RL implementations. The second research question focuses on structuring and organizing the identified code smells. By analyzing similarities and recurring patterns, this work aims to define meaningful categories of RL code smells. Finally, the third research question examines the relationship between code smell prevalence and project characteristics, such as project size, contributor count, and repository age.

4.4. Methodology

To address the research questions, we propose a two-phase methodology. In the first phase, we conduct a manual inspection of RL projects. In the second phase, we implement a tool to automatically detect the identified types of RL code smells in previously unseen projects.

4.4.1. First Phase: Manual Analysis of Projects

First, we mined open-source reinforcement learning projects from GitHub to collect relevant data for analysis. The source code of these projects was manually examined to identify and categorize RL-specific code smells.

Preliminary Analysis

For the preliminary analysis, we manually gathered 10 open-source projects from GitHub based on the following criteria:

- Relatively new projects: the last commit should not be older than three years. This ensures that the chosen repositories are up-to-date with the latest standards and advances in RL. Older projects may rely on deprecated libraries, outdated APIs, which could negatively impact the results, without providing any relevance to the current RL development.
- Reasonable number of stars on the repository: at least 50 stars on GitHub. GitHub stars are used as a metric for project visibility and popularity. Projects with more stars are more likely to represent solid RL development projects, excluding experimental or abandoned repositories. This increases the chance that the detected code smells in the result actually represent real life problems in the contemporary RL systems.
- Implementation of the projects in Python programming language. Python was selected because it is the prevalent programming language in RL research and industry applications.
- Use of reinforcement learning. We only selected projects that make use of RL libraries or frameworks, ensuring that the selected repositories implement RL logic.

We applied this selection criteria consistently in order to build a solid and meaningful dataset for our analysis. The intention was to focus on repositories that are actively maintained and recognized within the reinforcement learning community, as these projects are more likely to reflect real development practices. By doing so, we aim to analyze systems that have a certain level of maturity and practical relevance. At the same time, the criteria help filter out small experimental or hobby projects that may not contain enough structural complexity for a reliable code smell analysis. However, in order to further strengthen methodological transparency, we must acknowledge potential threats to validity:

- Selection bias: manual selection of projects may introduce researcher bias. Although predefined criteria were applied, the sample size is limited.
- Programming language restriction: limiting the analysis to Python projects improves consistency but reduces the generalizability to RL systems implemented in other programming languages.

4.4.2. Reinforcement Learning Code Smell Categories

During the manual inspection of reinforcement learning repositories, several recurring implementation issues were identified. These issues can be grouped into four broad categories depending on which component of the RL pipeline they affect. The categories are as follows:

Hyperparemeter-related code smells, such as the absence of hyperparemeter tuning of hard-coded hyperparameters, these issues can interfere with the reproducibility of the training.

Training-related code smells affect the learning process itself. Examples include the absence of hyperparameter tuning, missing checkpoint saving mechanisms, or the use of non-modular training loops. These issues reduce experiment reproducibility and may negatively influence training stability.

Environment-related code smells occur when the interaction between the agent and the environment is not handled correctly. Examples include redundant environment creation, missing environment termination, or lack of environment validation. These issues may lead to resource leaks or inconsistent simulation behavior.

Agent-related code smells relate to the logic used by the agent to interact with the environment. Examples include random agent behavior without proper exploration strategies, inconsistent reward processing, or ignoring key outputs returned by the environment such as rewards or termination signals.

The table 4.1 summarizes the RL specific code smells identified during the manual analysis phase.

Analyze the Source Code

For our initial manual analysis, we opted not to use static code analysis tools for Python, such as SonarQube [S32] or Pylint [S31], as they are not tailored to our context and

cannot account for the reinforcement learning aspects of the code. Instead, we conducted a detailed manual examination of the selected GitHub projects, which was done by systematically reviewing training scripts, environment definitions, agent implementations, test and example usage files, and experiment configuration files within each repository. Particular attention was given to the training loop, environment initialization, reward handling, evaluation procedures, and hyperparameter configuration. Each potential code smell was documented together with its location, context, and frequency across projects. To reduce personal subjectivity, identified issues were validated against reinforcement learning best practices, such as hyperparameter tuning [6] to ensure accuracy of detected code smells. Through this inspection, we identified 13 distinct code smells, having around 69 occurrences within the code. The identified RL-specific code smells are:

Manually Identified Code Smells	**No. of Occurrences**
No hyperparameter tuning implemented	31
No checkpoint saving implemented	7
Missing evaluation (agent or environment)	7
Missing termination/closing of the environment	6
Redundant environment creation	4
Inconsistent reward calculation	3
Tight coupling between components	2
Training and evaluation workflow coupling	2
Random agent behavior	2
Episode termination based on observation or reward	2
Ambiguous default logic for multi-agent setup	1
Ignored rewards, observations, or termination values	1
Non-modular control loop	1

Table 4.1.: Manually identified RL-specific code smells and their frequency

To provide a clearer understanding of the research work and the types of code smells we aim to identify in RL systems, we will closely examine two RL projects from the study and highlight the RL-specific code smells discovered.

SumoRL

SumoRL [S15] is a RL system that provides an easy-to-use interface to work with a reinforcement learning environment for Traffic Signal Control. It uses the Gymnasium [33] and the PettingZoo libraries [S16].

Taking a look at a code snippet in order to better illustrate RL-specific code smells, considering the following observations:

- Code snippet from: SumoRL [S21]

- irrelevant imports were removed for the sake of keeping the script shorter

```python
# Code snippet from:
# https://github.com/LucasAlegre/sumo-rl/blob/main/experiments/dqn_big-intersection.py

import gymnasium as gym
from stable_baselines3.dqn.dqn import DQN
import numpy as np
import traci

from sumo_rl import SumoEnvironment

env = SumoEnvironment(
    net_file="sumo_rl/nets/big-intersection/big-intersection.net.xml",
    single_agent=True,
    route_file="sumo_rl/nets/big-intersection/routes.rou.xml",
    out_csv_name="outputs/big-intersection/dqn",
    use_gui=True,
    num_seconds=5400,
    yellow_time=4,
    min_green=5,
    max_green=60,
)

model = DQN(
    env=env,
    policy="MlpPolicy",
    learning_rate=1e-3,
    learning_starts=0,
    buffer_size=50000,
    train_freq=1,
    target_update_interval=500,
    exploration_fraction=0.05,
    exploration_final_eps=0.01,
    verbose=1,
)

model.learn(total_timesteps=100000)
```

Listing 4.1: Example RL script from SumoRL

What we can identify in the above script are the following:

- A hyperparameter tuning mechanism is missing. The script specifies multiple critical hyperparameters (e.g., learning rate, buffer size, exploration rate) directly within the model initialization, without employing any tuning or optimization strategy. In reinforcement learning, performance is highly sensitive to hyperparameter selection. The absence of systematic tuning (e.g., grid search [20], Bayesian optimization [11], or Optuna-based [S4] search) can significantly limit reproducibility and model performance. This issue is especially critical in RL systems where training instability is common.

- No checkpoint saving implemented: the script does not implement intermediate saving of the models. Therefore, errors can lead to increased resource costs.
- Missing termination or closing of the environment: not closing the environment could lead to resource leaks or unpredictable behavior in subsequent runs. This is the recommended way to [7] close the environment: `env.close()`
- Missing evaluation (agent or environment): Without validation, there is no way to assess the performance of the trained agent.

In general, the code smells identified in SumoRL mainly affect training stability and reproducibility. The absence of checkpoint saving and evaluation mechanisms reduces experimental reliability, while missing environment cleanup may lead to resource management issues. These issues suggest that although the implementation is functional, it lacks certain engineering practices necessary for large-scale or long-running RL experimentation.

SMARTS

SMARTS [42], which stands for Scalable Multi-Agent Reinforcement Learning Training School, is a simulation platform designed for multi-agent reinforcement learning (RL) and autonomous driving research. It focuses on creating realistic and diverse interactions between agents and is part of the XingTian suite of RL platforms developed by Huawei Noah's Ark Lab.

Applying the same technique as before, we studied the implementations in the library that illustrate a multi-agent interaction and identify specific RL code smells present.

Observations:

- Code snippet from SMARTS [S22]
- this snippet represents only a subsection of the full implementation.

```
N_AGENTS = 4
AGENT_IDS: Final[list] = ["Agent %i" % i for i in range(N_AGENTS)]


class RandomLanerAgent(Agent):
    def __init__(self, action_space) -> None:
        self._action_space = action_space

    def act(self, obs, **kwargs):
        return self._action_space.sample()


class KeepLaneAgent(Agent):
    def __init__(self, action_space) -> None:
        self._action_space = action_space

    def act(self, obs, **kwargs):
        return self._action_space.sample()
```

```
19
20
21 def main(scenarios, headless, num_episodes, max_episode_steps=None):
22
23     # This interface must match the action returned by the agent
24     agent_interfaces = {
25         agent_id: AgentInterface.from_type(
26             AgentType.Laner,
27             max_episode_steps=max_episode_steps
28         )
29         for agent_id in AGENT_IDS
30     }
31
32     env = gym.make(
33         "smarts.env:hiway-v1",
34         scenarios=scenarios,
35         agent_interfaces=agent_interfaces,
36         headless=headless,
37     )
38
39     for episode in episodes(n=num_episodes):
40
41         agents = {
42             agent_id: RandomLanerAgent(env.action_space[agent_id])
43             for agent_id in agent_interfaces.keys()
44         }
45
46         observations, _ = env.reset()
47         episode.record_scenario(env.unwrapped.scenario_log)
48
49         terminateds = {"__all__": False}
50
51         while not terminateds["__all__"]:
52
53             actions = {
54                 agent_id: agent.act(observations)
55                 for agent_id, agent in agents.items()
56             }
57
58             observations, rewards, terminateds, truncateds, infos = env.
    step(actions)
59
60             episode.record_step(
61                 observations,
62                 rewards,
63                 terminateds,
64                 truncateds,
65                 infos
66             )
67
68     env.close()
69
70
```

```
if __name__ == "__main__":

    parser = minimal_argument_parser(Path(__file__).stem)
    args = parser.parse_args()

    if not args.scenarios:
        args.scenarios = [
            str(SMARTS_REPO_PATH / "scenarios" / "sumo" / "loop"),
        ]

    build_scenarios(scenarios=args.scenarios)

    main(
        scenarios=args.scenarios,
        headless=args.headless,
        num_episodes=args.episodes,
        max_episode_steps=args.max_episode_steps,
    )
```

Listing 4.2: Example multi-agent environment interaction script

Some of the code smells we identified in this project are as follows:

- Once again, no hyperparameter tuning has been implemented in the code; tuning is an important step in the RL pipeline; in order to obtain the best training and learning results, we must find optimal values for the hyperparameters. [6]
- The reward [4] is an essential concept in RL because the agent learns how to make good actions over time based on the rewards it gets. The agent should also try to get the maximum reward over time. In the above code snippet, the reward variable is retrieved from the environment but is never processed or checked:

```
observations, rewards, terminateds, truncateds, infos = env.step(
    actions)
episode.record_step(observations, rewards, terminateds, truncateds,
    infos)
```

- No environment validation: after the environment is created, there are no specific validations to confirm proper creation. In reinforcement learning systems, environment validation is important to ensure that the simulation environment is correctly initialized, conforms to expected interface specifications, and is capable of producing consistent observations and rewards. Without validation checks, runtime errors or incorrect environment configurations may remain undetected until training is already in progress. This issue is particularly relevant in RL systems because training processes are often computationally expensive and detecting early misconfiguration of the environment can significantly improve experimental efficiency and reproducibility. [22]

In the SMARTS example, the identified smells mainly affect learning quality and experimental validity. Ignoring reward processing undermines the core learning signal of

the agent, while the absence of hyperparameter tuning limits potential for optimizations. These findings highlight how RL-specific implementation choices directly influence learning effectiveness.

In conclusion of the first phase, the examples above illustrate recurring RL-specific code smells related to training configuration, evaluation, and reward handling. These observations support **RQ1** by demonstrating that certain smells, particularly missing hyperparameter tuning and evaluation mechanisms, occur repeatedly across projects. Furthermore, they provide a foundation for **RQ2**, as identified issues can be grouped into broader categories such as training-related, environment-related, and agent-related code smells.

4.4.3. Second Phase: Static Reinforcement Learning Code Analysis Tool

In order to reduce the manual effort required for identifying reinforcement learning code smells, a static analysis tool was developed to automatically detect defined RL specific smells. The tool analyzes open-source RL repositories and classifies detected issues according to the predefined categorization, while also providing aggregated statistics on the distribution of smell categories.

In the second phase of the study, a total of 51 reinforcement learning projects were analyzed using the developed tool. The automated analysis identified 6392 heuristics, out of which 4,872 were classified as code smell occurrences, which is significantly higher than the number of smells detected during the preliminary manual inspection. This difference is expected, since the automated tool enables comprehensive scanning of repository codebases and enables us to analyze and study more projects, constructing a solid data set.

An additional advantage of the developed tool is its ability to generalize to previously unseen repositories, allowing code smell detection beyond the initially analyzed projects. This supports further experimentation and analysis within the scope of this research. Further technical details regarding the design, implementation, and functionality of the RL code smell detection tool are presented in the following chapter.

5. RL code smells detector

This project presents a static analysis tool for detecting code quality issues in RL Python projects. The tool identifies anti-patterns and potential issues that may negatively affect training performance, maintainability, and reproducibility.

5.1. Concepts

5.1.1. RL Code Smells

Code smells are indicators of potential problems in source code that may lead to larger design or maintainability issues over time. Although they are not bugs and do not directly affect the functionality of the software, they often point to weak design decisions, poor coding practices, or code structures that become difficult to maintain and extend [30]. Common examples of code smells include long methods, large classes, duplicated code, excessive parameters, dead code, and the use of hard-coded values such as magic numbers or strings. These issues can increase the complexity of the codebase, make the code harder to understand, and contribute to technical debt and future software defects [30].

In this thesis, when talking about RL code smell, we refer to code smells that are specific to reinforcement learning applications. The following code smell categories are defined based on the RL concept in which they appear, one thing to note is that we also introduced a CODESTYLE category, which is also applicable for applications in other domains as well:

- HYPERPARAMETER category
- TRAINING category
- AGENT category
- ENVIRONMENT category
- EVALUATION category
- CODESTYLE category

In the following subsections, we explore each smell category and go into further details on what code smells are detected by the tool belonging to each category and why we consider this a "RL specific code smell".

5.1.2. HYPERPARAMETER Code Smell Category

In this category, we look for two types of code smells that involve hyperparameters, in the RL pipeline:

- **Hard-coded hyperparameters**: a hyperparameter is assigned a constant value directly in the code (the tool considers most common hyperparameters like: learning rate, gamma, discount factor, epsilon, exploration rate, target update interval, decay steps, max steps, batch size, reward). Why do we consider this a "RL specific code smell"?
  - **Reproducibility**: hard-coded hyperparameters in the code are prone to be modified by mistake and are more difficult to track across experiments runs. A cleaner solution would be providing configuration files with hyperparameter values, so that those values are centralized in one place, making experiment runs easy to reproduce.
  - **Lack of tuning evidence**: the presence of fixed hyperparameter values often suggests that parameters were selected manually or arbitrarily, without a documented optimization process. In reinforcement learning, performance is highly sensitive to hyperparameters such as `learning_rate`, `gamma`, or `batch_size`, making proper tuning an essential step for achieving stable and reliable results.
  - **Maintainability**: hard-coded hyperparameters may appear in multiple locations within the codebase. This duplication increases the risk of inconsistencies if a value is updated in one place but not in others, potentially leading to unintended behavior or configuration errors.
- **No tuning of hyperparameters**: no hyperparameter tuning library is imported or called anywhere in the script. Tuning libraries that are recognized by the tool are the following: optuna, ray.tune, sklearn, hyperopt, bayes opt. We categorize this as a RL specific code smells because of the following reasons [6]:
  - **Sensitivity to hyperparameters:** RL algorithms are highly sensitive to the choice of hyperparameters, even small configuration changes in parameters such as the learning rate or discount factor (γ) can significantly affect training performance and may determine whether the agent converges to a good policy or not [15].
  - **Lack of fixed training data:** unlike supervised learning, RL systems generate training data through interactions with the environment, which makes it difficult to determine suitable hyperparameter values in advance. Therefore, finding the best values for the hyperparameters is crucial in order to have good training results.
  - **Environment-specific behavior:** hyperparameter values that perform well in one environment may not be suitable for other environments, since they can be highly dependent on the environment and the domain.

5.1.3. TRAINING Code Smell Category

- **Training and evaluation coupling**: for example, when training function calls `evaluate()`, uses `EvalCallback`. We consider this case a RL specific code smell because of the following reasons:
  - **Exploration vs. exploitation trade off**: training requires exploration (stochastic policy, epsilon-greedy, noise injection). Evaluation should use the deterministic/greedy policy. When they are coupled, there's a risk that the evaluation runs with exploration active, resulting in misleading performance numbers.
  - **"Biased" results**: when the evaluation is done on the same environment the agent just trained on, it may appear to perform well simply because it has adapted to that specific instance.
- **Missing checkpoint saving**: this occurs when there are no intermediate checkpoints saved during training. We classify this as a training code smell because:
  - **Loss of progress**: progress might be completely lost during training if an interruption or an error occurs, which would require for the whole training process to be re-started from scratch.
  - **Difficult debugging**: for example, in case the agent starts to behave unexpectedly at some step in the training, checkpoints can help with debugging, and without checkpoints this becomes nearly impossible.

5.1.4. AGENT Code Smell Category

The following code smells were identified as common issues related to agent behavior and decision-making logic in reinforcement learning systems:

- **Random action selection**: This occurs when an agent's `act()` method directly returns `action_space.sample()` instead of consulting a learned policy. This is problematic because the `act()` method represents the core decision-making component of the agent. If actions are always selected randomly, the agent does not learn from experience. This pattern often indicates placeholder or incomplete implementations, where a proper policy has not yet been integrated.
- **Random behavior through sampling**: In this case, `action_space.sample()` is used within the codebase without any evidence of policy-based action selection methods (e.g., `predict`, `act`, `choose_action`, or `select_action`). In a correct RL implementation, the agent should rely on its learned policy to determine actions. The presence of random sampling without policy usage suggests that the training loop is incomplete or that the policy is never actually invoked.
- **Invalid action execution in multi-agent settings**: This occurs when the environment is advanced using a call such as `env.step({})`, where an empty action

dictionary is provided. In multi-agent environments, actions are typically passed as dictionaries mapping agents to their actions. Supplying an empty dictionary means that no agent is assigned an action, leading to meaningless environment transitions. This often indicates incomplete implementation of the action selection logic.

- **Ambiguous multi-agent initialization**: This smell arises when multi-agent configuration flags and agent count variables are present but lack a clear or consistent initialization pattern. In multi-agent RL systems, correct initialization of agents and their roles is essential. Ambiguity in this process can lead to missing agents, incorrect policy assignments, or mismatches between action and observation spaces. Such issues are often difficult to detect early and may only surface during training.

5.1.5. ENVIRONMENT Code Smell Category

The following code smells were identified as common issues related to environment management and lifecycle handling in reinforcement learning systems:

- **Redundant environment creation**: This occurs when a training environment is instantiated normally, and then a second instance of the same environment is created only for auxiliary purposes, such as video recording (e.g., using `VecVideoRecorder`). This practice is inefficient because RL environments can be computationally expensive to initialize, often involving physics simulators, rendering engines, or complex reward mechanisms. Creating duplicate environments increases memory and computational overhead. Furthermore, it introduces the risk of inconsistency, as the recording environment may differ in configuration (e.g., random seeds, wrappers), resulting in recorded episodes that are not representative of the actual training process. A more appropriate approach is to wrap the existing environment conditionally.

- **Environment closure handling**: This pattern is identified when the method `env.close()` is invoked. While not inherently a code smell, it is treated as an informational warning due to its potential to introduce subtle errors. In RL systems, calling `env.close()` prematurely-such as within the training loop or before all interactions are completed-can terminate the underlying simulation unexpectedly. This may lead to runtime failures or undefined behavior in subsequent calls to `env.step()` or `env.reset()`. Therefore, it is important to ensure that environment closure occurs only at the appropriate point, typically at the end of execution.

5.1.6. EVALUATION Code Smell Category

The following code smells were identified as common issues related to model evaluation and validation in reinforcement learning systems:

- **Missing model evaluation**: This occurs when no evaluation-related calls are present in the codebase (e.g., `model.eval()`, `model.evaluate()`, `EvalCallback`,

or the use of dedicated test or validation environments). This is a critical issue in reinforcement learning, as the training reward is often an unreliable indicator of true agent performance. Training rewards are influenced by the exploration strategy and may not accurately reflect the quality of the learned policy. Without a separate evaluation phase-typically involving a deterministic or greedy policy executed on independent episodes-it is not possible to reliably assess whether the agent has learned an effective strategy. The absence of evaluation raises significant concerns regarding the validity and reproducibility of experimental results.

5.1.7. CODESTYLE Code Smell Category

In this category, we only detect code smells that can impact the monitoring and observability in reinforcement learning systems.

- **Missing logging**: This occurs when no logging libraries (e.g., `logging`, `loguru`, `stable_baselines3.common.logger`, `ray.tune.logger`) are imported and no logging calls (such as `info`, `debug`, `warning`, `error`, or `critical`) are present in the code. This is particularly problematic in reinforcement learning, where training is typically a long-running and non-transparent process. Without logging, there is no visibility into the internal state of training. Important signals such as episode rewards, loss values, entropy, learning rate schedules, environment resets, or exploration parameters cannot be monitored. As a result, issues such as training divergence, reward exploitation, or stalled learning may go undetected until the end of execution, potentially wasting significant computational resources.

5.2. Key Features

The RL code smell detector tool has the following integrated features and relies on them for a good detection process:

- **RL framework detection:** Recognizes `gym`, `stable_baselines3`, `sb3_contrib`, `ray.rllib`, `rlberry`, `torchrl`, and torch
- **Smart pre-filtering:** analyzes only files containing RL imports and environment usage.
- **GitHub integration:** clones repositories and retrieves metadata such as repository age, tags, number of contributors.
- **Interactive visualization:** Provides Plotly-based bar charts with category filtering.
- **Export options:** generates CSV reports (complete and category-based breakdown).
- **Line-level detection:** reports include exact line numbers for each detected issue.

5.3. Abstract Syntax Tree in Python

AST is a module in the Python standard library. Python code is converted to an abstract syntax tree before being transformed into bytecode(.pyc files). One of the most important functions of the AST python module is generating the tree, which is an internal representation that reflects the syntactic function of the python file [S24]. The AST provides a hierarchical tree-based model of the source code, this serves multiple purposes, such as enabling analysis, transformation, and compilation [25]. In AST each node corresponds to a syntactic construct in the program, but in the AST symbols and characters such as parentheses and formatting are removed, focusing instead on structural relationships between language constructs. It is used both internally by the interpreter and externally by development tools for static analysis and code transformation [25, 26]. [S24]

We chose to use Python's AST module because it provides a structured representation of source code that captures its syntactic and hierarchical structure, which makes it particularly well suited for identifying code smells because those are typically defined in terms of structural patterns rather than textual characteristics [25], and is well suited for python projects, which is primarily used in RL applications, therefore another reason for which we only analyze Python based projects.

The AST is constructed after lexical analysis and parsing, and before compilation into bytecode. The process can be summarized as follows:

- **Lexical analysis:** the source code is converted into a sequence of tokens representing syntactic elements.
- **Parsing:** tokens are analyzed according to Python's grammar rules to determine syntactic correctness and structure [26].
- **AST construction:** the parsed structure is transformed into an Abstract Syntax Tree [25].
- **Compilation:** the AST is compiled into bytecode executed by the Python virtual machine

The AST consists of nodes representing syntactic constructs. These nodes form a hierarchical structure reflecting the nested nature of programming languages. Each node contains information about its type and relationships to child nodes. The AST abstracts away concrete syntax and instead represents logical relationships between constructs such as expressions, statements, and operators [25].

Understanding the structure of an AST node is crucial for working with ASTs effectively. Each node represents a specific element of the program's syntax, such as functions, statements, and expressions. AST nodes have a defined structure, consisting of a type, attributes, and child nodes [S25]. An AST node has the following structure:

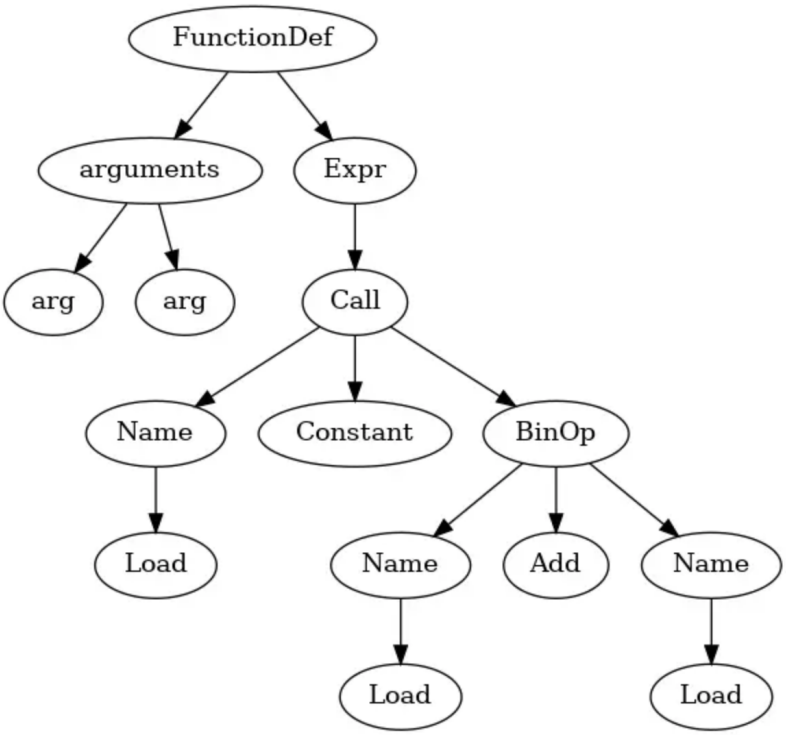

Figure 5.1.: AST tree (from [S25])

Since code smells can be labeled as recurring structural patterns within a program, with the help of the AST, we can do systematic traversal of code elements, which allows the identification of code smell such as excessive nesting, overly large functions, and other potential code issues. AST supports rule-based detection mechanisms that operate on node types and their relationships, because it explicitly represents program structure. One key advantage of AST-based analysis is its semantic awareness.

Since the AST is built directly from Python's formal grammar, every structure it captures is guaranteed to be syntactically valid. This alone makes it more reliable than simple string matching or regex-based approaches, which are prone to false positives. On top of that, AST nodes carry contextual information, so it is straightforward to tell apart things like function calls, variable assignments, and control flow constructs [26].

What makes the AST particularly useful for this work is that it exposes structural metrics that are commonly linked to code quality. For example, it is easy to count the number of statements in a function, measure how deeply control structures are nested, or track how often certain node types appear. These kinds of metrics are widely used in static analysis tools, and being able to traverse the tree programmatically makes it easy to build flexible, extensible analyses on top of them [S25].

5.4. Code Smell Detection Process

The RL code smell detection tool follows a structured, multi-stage static analysis pipeline. The pipeline is designed to automatically ingest, filter, analyze, and finally construct reports with the findings, which consist of RL code heuristics from open-source repositories. The overall process consists of seven sequential steps, described below.

5.4.1. Step 1 - Repository Ingestion

The entry point of the application is the repository ingestion. The user must enter a GitHub url of the project to be analyzed. The application clones the repository into a temporary local directory using the `git` commands, therefore the repository must be open source and publicly available.

In parallel, we also gather repository-level metadata via the GitHub API [S17], that we consider relevant for future the case study of the repository, this includes attributes such as the number of stars, contributor count, repository size, associated topics and tags, and repository age (computed as the time difference between the first commit and the current date, expressed in months). This metadata is displayed alongside the analysis results for future correlations analysis in the research but it is not used during the code smell detection process.

5.4.2. Step 2 - File Discovery (ProjectReader)

After the repository is cloned locally, the `ProjectReader` component performs a recursive traversal of the project directory to identify relevant source files. We only want to analyze files and scripts that are RL relevant, therefore, firstly we gather all Python files (e.g., files with the `.py` extension) and then filter them based on the following criteria:

- Files located within `/venv/` directories are excluded, as these represent external dependencies rather than project-specific code.
- Files prefixed with `._` are excluded, as they correspond to metadata artifacts.

The output of this step is a flattened list of Python source files that serve as candidates for further analysis.

5.4.3. Step 3 - RL Script Filtering (RLScriptDetector)

In order to only analyze files that are contain RL specific logic, each discovered file is evaluated to determine whether it represents a reinforcement learning script. This filtering is performed by parsing the file into an AST [S18] and search for:

- **RL**: imports of known RL libraries (e.g., `gym`, `stable_baselines3`, `ray.rllib`, `torch`), instantiations of RL algorithms (e.g., PPO, DQN, A2C, SAC, TD3), or agent interactions, defined by calls to common RL methods such as `.step()`, `.predict()`, `.learn()`, and `.train()`

- **Environment-related code**:environment instantiations (`gym.make()`), variables containing the term `env`, class definitions containing `Env`, or factory functions such as `make_env` or `create_env`

In the end, a file is marked as a RL script only if both conditions are true, this ensures that non-RL files (such as utilities or configuration files without any RL context) are excluded from further analysis.

5.4.4. Step 4 - AST Parsing and Analysis (Analyzer)

In the next step, the discovered RL files are passed in the next stage of the pipeline: parsing them into an AST [S18], which provides a structured, execution-independent representation of the source code.

The `Analyzer` component traverses the AST using a visitor pattern [S19]. For each encountered node, the analyzer sends over the node to all relevant detectors for the node. This enables multiple detectors to operate concurrently over the same traversal.

```
class Analyzer(ast.NodeVisitor):
    def __init__(self):
        self.environment_smells = EnvironmentSmellsDetector()
        self.checkpoint_smells = CheckpointSmellsDetector()
        self.hyperparameter_smells = HyperparametersSmellsDetector()
        self.evaluation_smells = EvaluationSmellsDetector()
        self.logging_smells = LoggingDetector()
        self.initialization_smells = InitializationSmellsDetector()
        self.agent_smells = AgentSmellsDetector()
        self.training_smells = TrainEvalCouplingDetector()
        self.report = []
```

Listing 5.1: Analyzer class initialization

AST Node Type	Invoked Detectors
`visit_Call`	Checkpoint, Hyperparameter, Evaluation, Environment, Logging, Agent, Training
`visit_Assign`	Hyperparameter, Evaluation, Environment, Initialization
`visit_Import / visit_ImportFrom`	Logging, Hyperparameter
`visit_FunctionDef`	Training
`visit_For / visit_While`	Checkpoint
`visit_If`	Initialization
`visit_Expr`	Environment
`visit_arguments`	Training

Table 5.1.: Mapping between AST node types and invoked detectors

Each detector keeps track of information while analyzing the code, allowing it to gather context before deciding whether a code smell is present.

5.4.5. Step 5 - Per-Detector Logic

In the next stage of the pipeline, each detector uses one of two detection approaches: the first is **immediate flagging**, where a smell is reported as soon as a suspicious pattern is found. For example, a standard hard-coded hyperparameter is found right at the beginning of the script, the detector can immediately flag it and mark it as code smell. The second approach is **deferred flagging**, where the detector collects information during the full AST traversal and only decides at the end whether a smell is present. For example, only after the whole RL script is analyzed we can say if there is an intermediary check point saving or not, or if the environment is properly closed or not, which, if missing, would be flagged as a code smell.

All detected heuristics are stored as `Heuristic` objects. These objects contain information such as the smell name, a short description, the line number, if applicable, a boolean flag indicating whether the issue is considered a real smell or an additional information (for example checkpoint saving was detected, this will be not be flagged as a code smell), and a category.

5.4.6. Step 6 - Report Aggregation

Finally, once the AST traversal is complete, the analyzer aggregates the results produced by all detectors into a unified list of `Heuristic` objects. This list is then encapsulated in a `Report` object associated with the analyzed file, and all file-level reports are subsequently combined into a global result set representing the repository as a whole.

5.4.7. Step 7 - Results Visualization

The aggregated results are transformed into a structured format using the Pandas library and presented through the Streamlit interface. The tool provides the following outputs:

- a detailed results table containing file names, detected smells, descriptions, line numbers, categories, and classification flags (figure 5.6)
- an interactive bar chart illustrating the distribution of detected code smells across categories. (figures 5.9 and 5.7)
- a summary table presenting category-level counts and percentages. (figure 5.8)
- downloadable CSV reports for both detailed results and aggregated statistics, together with repository metadata. (figure 5.9)

5.4.8. Technologies and Libraries

Table 5.2 summarizes the main libraries used in the implementation.

Library	Version	Purpose
streamlit	∼1.49.1	Web UI framework
pandas	∼2.3.2	Data manipulation and report generation
plotly	∼6.3.0	Interactive visualizations
openai	∼0.28.0	LLM integration (currently not used; reserved for future use)
pymongo	4.10.1	MongoDB driver (potential result storage backend)
dnspython	2.7.0	DNS utilities

Table 5.2.: Technologies and libraries used in the implementation.

- **Streamlit** [S5] is used as the primary framework for building the web-based user interface. It enables rapid development of interactive dashboards, very fast deployments, and allows the analysis results to be visualized directly in the browser without requiring complex front-end development.
- **Pandas** [S6] is used for data manipulation and processing. It is used to structure the results generated by the code smell detector, aggregate statistics, and prepare data for visualization and reporting.
- **Plotly** [S7] is used to generate interactive visualizations. It allows users to explore the distribution of detected code smells and their categories through dynamic charts and graphs within the web interface.
- **PyMongo** [S8] provides the interface for interacting with MongoDB databases. It is included as a potential backend solution for storing analysis results and metadata related to processed repositories.
- **dnspython** [S9] it was introduces as a supporting dependency for network-related operations required by the database connectivity layer.

Other considerations:

- **OpenAI API** [S10] was initially considered for integrating Large Language Model (LLM) capabilities to assist with automated interpretation of detected code smells. To be considered in potential of future work.

Additionally, several built-in Python modules are used throughout the implementation, including `ast` for static code parsing, `subprocess` for executing system-level operations, `tempfile` for managing temporary files, `re` for regular expression processing, `os` for operating system interactions, `csv` and `json` for data serialization, `requests` for HTTP communication, and `datetime` for time-related operations.

5.5. Project Structure

The overall project structure is organized as follows:

```
rl-code-smell-detector/
|-- app.py                  # Streamlit web interface
|-- main.py                 # CLI entry point
|-- analyzer.py             # Main analysis orchestrator
|-- pre_processing.py       # RL script detection
|-- project_reader.py       # Files utilities functions
|-- github_utils.py         # GitHub API integration
|-- cli_utils.py            # CLI display and reporting
|-- ui_utils.py             # UI data transformations
|-- model/
|   |-- category.py         # Code smell categories enum
|   |-- heuristic.py        # Detection unit model
|   |-- report.py           # Report aggregation model
|-- detectors               # package containing eight specialized detectors
```

5.6. Architecture

One key architectural design decision of the tool is its reliance on pure static analysis. The detection process operates exclusively on the AST representation of the source code and the implemented code smell detectors, without executing any code, instantiating environments, or loading models. An advantage of this approach is that it ensures that the tool is safe, fast, and applicable to arbitrary repositories regardless of their dependencies or runtime requirements, however, on the other side, this method also has its own limitations, as only syntactically observable patterns can be detected and dynamic behaviors such as runtime configuration loading may go undetected.

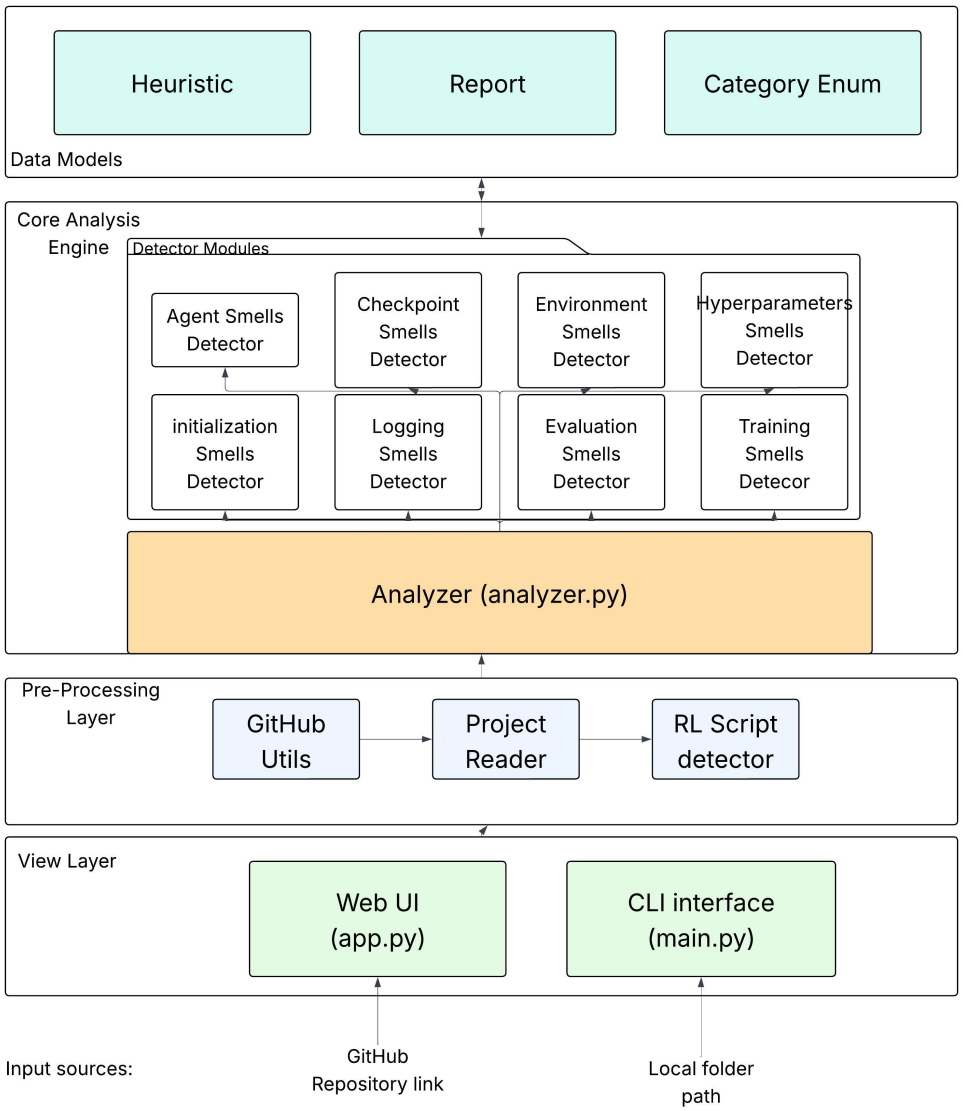

Figure 5.2.: Component architecture of the RL Code Smell Detector.

The core analysis engine is built on the Visitor design pattern [S19], implemented using Python's `ast.NodeVisitor`. The central `Analyzer` class orchestrates eight specialized detector classes, focusing on a distinct category of code smells, more exactly: hyperparameter issues, missing checkpoints, improper environment handling, missing evaluation routines, agent-related issues, initialization problems, insufficient logging, and training-evaluation coupling. Each detector produces instances of the `Heuristic` model, which captures a descriptive name, detailed information about the finding, the corresponding line number in the source code, a boolean flag indicating whether the finding constitutes a

code smell, and an assigned category. These heuristics are aggregated into `Report` objects that summarize the analysis results for each examined file.

5.7. Architectural Design Decisions

This section explains the main architectural decisions behind the RL Code Smell Detector and why they are suitable for a static analysis tool focused on reinforcement learning projects.

5.7.1. Pipeline Architecture

The system follows a sequential pipeline structure:

$$\text{Input} \rightarrow \text{Repository cloning} \rightarrow \text{Filtering} \rightarrow \text{Parsing} \rightarrow \text{Detection} \rightarrow \text{Reporting} \rightarrow \text{Results visualization}$$

This structure naturally reflects how static code analysis works. Each stage has a clearly defined responsibility and well-defined input and output. This makes the system easy to understand and debug. If an issue occurs, it is possible to trace it back to a specific stage in the pipeline, which improves maintainability and transparency.

5.7.2. Visitor Pattern for AST Traversal

The core analysis is implemented using Python's `ast.NodeVisitor`, which is the standard approach for traversing abstract syntax trees in Python.

This provides several advantages:

- **Separation of concerns:** The AST library handles tree traversal, while detection logic is implemented in dedicated `visit_*` methods.
- **Structured dispatching:** Each node type is handled by a specific method, avoiding manual type checks and improving readability.
- **Extensibility:** Supporting additional node types only requires implementing a new `visit_*` method.

5.7.3. Strategy Pattern with Specialized Detectors

Regarding the implementation details of the tool, each detector is implemented in a specialized class, instead of having the whole detection algorithm in one large analyzer class. Each detector is responsible for one specific category of code smells (e.g., hyperparameters, training logic, environment handling).

This design offers the following advantages:

Advantage	Impact
Single Responsibility	Each detector focuses on exactly one smell category, improving clarity and maintainability.
Independent Testing	Detectors can be unit tested in isolation.
Parallel Development	Different detectors can be implemented or improved independently.
Easy Extension	Adding a new smell category only requires adding a new detector class.
Selective Execution	Specific detectors can be enabled or disabled without affecting the others.

Table 5.3.: Advantages of using specialized detector classes.

As already mentioned, the system implements eight specialized detectors, each encapsulating a dedicated detection strategy:

- **AgentSmellsDetector**: detects random action selection and policy-related issues.
- **CheckpointSmellsDetector**: identifies missing checkpoint saving.
- **EnvironmentSmellsDetector**: detects unclosed or improperly managed environments.
- **EvaluationSmellsDetector**: identifies missing evaluation routines.
- **HyperparametersSmellsDetector**: detects hard-coded hyperparameters.
- **InitializationSmellsDetector**: identifies multi-agent setup issues.
- **LoggingDetector**: analyzes logging practices.
- **TrainingSmellsDetector**: detects training $\leftrightarrow$ evaluation coupling problems.

5.7.4. Composite Pattern

The `Analyzer` class aggregates all detectors and exposes a single method, `get_report()`. External components (CLI or Web UI) do not need to interact with individual detectors. This reduces the coupling between the user interface and the analysis logic and provides a clean abstraction layer.

5.7.5. Pre-Filtering with RLScriptDetector

Before performing AST analysis, files are filtered using the `RLScriptDetector`. Only Python scripts that contain reinforcement learning related imports or patterns are analyzed further.

This improves performance by avoiding unnecessary processing of unrelated files. It follows the fail-fast principle: invalid or irrelevant inputs are rejected as early as possible.

5.7.6. Suitability for Reinforcement Learning Projects

The architecture is particularly well suited for RL specific analysis because reinforcement learning systems typically contain clearly separated concern areas such as hyperparameters, environments, agents, training loops, and evaluation logic. The modular detector structure directly reflects these domains. Additionally, new experimental heuristics can be added safely without modifying existing and validated components, which makes the system suitable for research-oriented extensions.

Overall, the architecture emphasizes modularity, extensibility, and maintainability. The clear separation of responsibilities ensures that the system remains scalable as additional detectors are introduced, thus making the architecture well suited for an evolving static analysis tool in the reinforcement learning domain.

5.7.7. Application structure

The internal structure of the application, described via a comprehensive UML diagram is shown in figure 5.3. Each smell detector is encapsulated in its own class, all of them being initialized and used in the Analyzer class. Among other worth mentioning classes are the models: the `Heuristic`, the `Report` and the `Category` enum.

```
class Heuristic:
    def __init__(self, name, details, line_nr, is_code_smell, category):
        self.name = name
        self.details = details
        self.line_nr = line_nr
        self.is_code_smell = is_code_smell
        self.category = category
```

```
class Report:
    def __init__(self, filename, heuristics: List[Heuristic]):
        self.filename = filename
        self.heuristics = heuristics
```

```
from enum import Enum

class Category(Enum):
    HYPERPARAMETER = 1
    TRAINING = 2
    AGENT = 3
    ENVIRONMENT = 4
    EVALUATION = 5
    CODESTYLE = 6
```

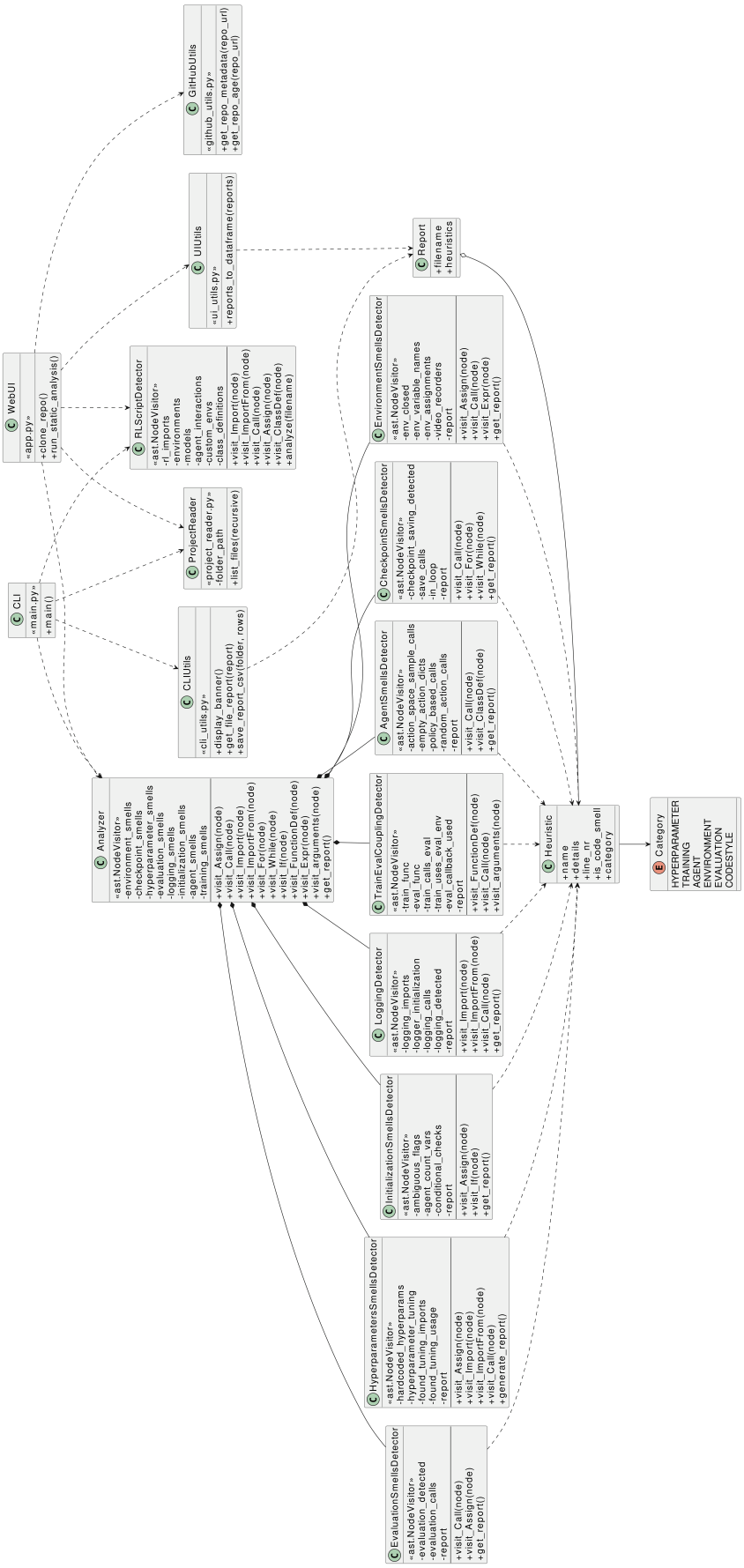

Figure 5.3.: UML diagram of RL Code Smell Detector.

5.8. User Interface

The tool supports two user interface modes:

- **CLI mode** (`main.py`): enables interactive analysis of local project folders.
- **Web UI mode** (`app.py`): a Streamlit-based web application to analyze GitHub repositories, including interactive visualizations.

5.8.1. Web UI Interface

The Web UI interface is rather simple and very intuitive. It first display a text box to enter the GitHub url of the project one wants to analyze and then simply clicking the button "Analyze Repository".

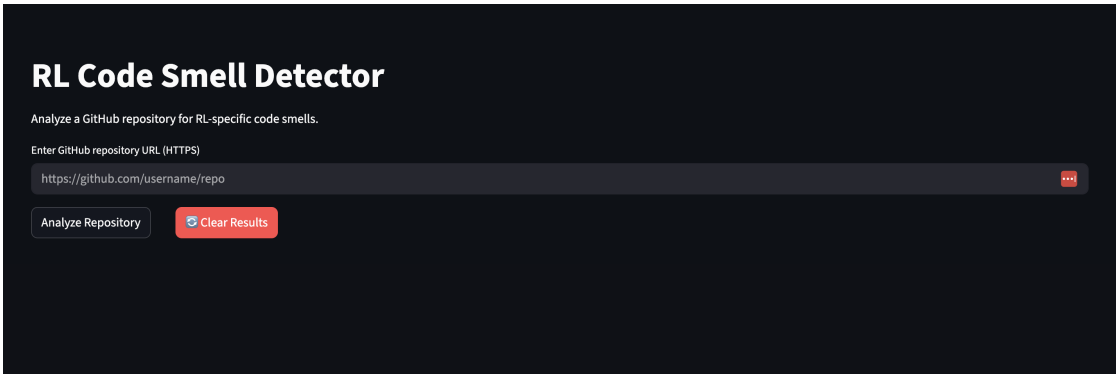

Figure 5.4.: Home page - web UI interface

After this, the application starts to analyze the repository, providing the user with constant feedback and showing the steps it covers, especially the pre-analyzing phase, and it also displays the metadata of the repository, as shown in the figure 5.5.

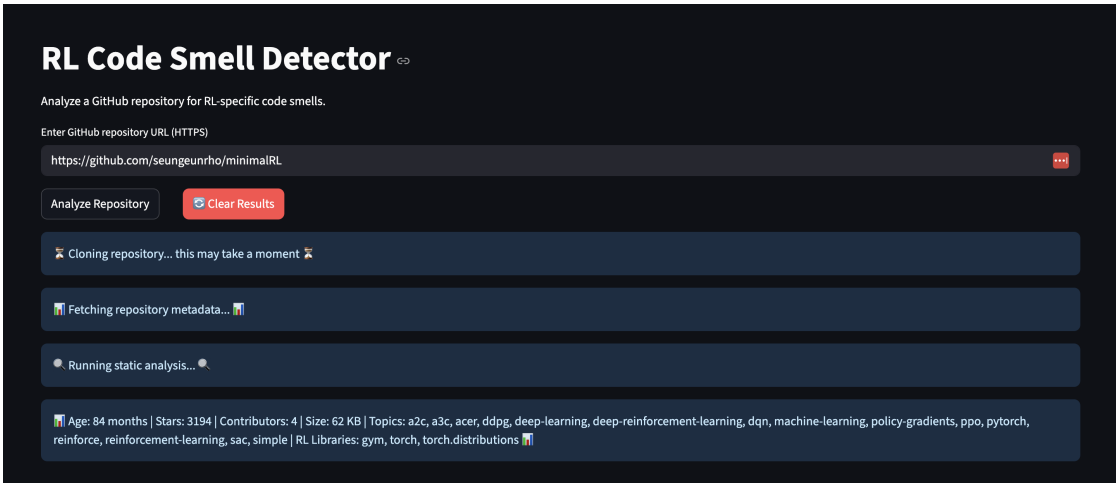

Figure 5.5.: Pre analysis step - web UI interface

The UI is a single page application. The results are shown on the same page, it will first print out a full report of the detected heuristics, not only detected code smells, shown in figure 5.6. The following section is the *Distribution of Code Smells by Category* (figure 5.8), which also offers the functionality of clickable bar charts, resulting in a report of the heuristics that are part of the chosen category (figure 5.9). Lastly, the tool shows the *Code Smell Category Breakdown* - with the possibility to download the overview category report in CSV format, in figure 5.7

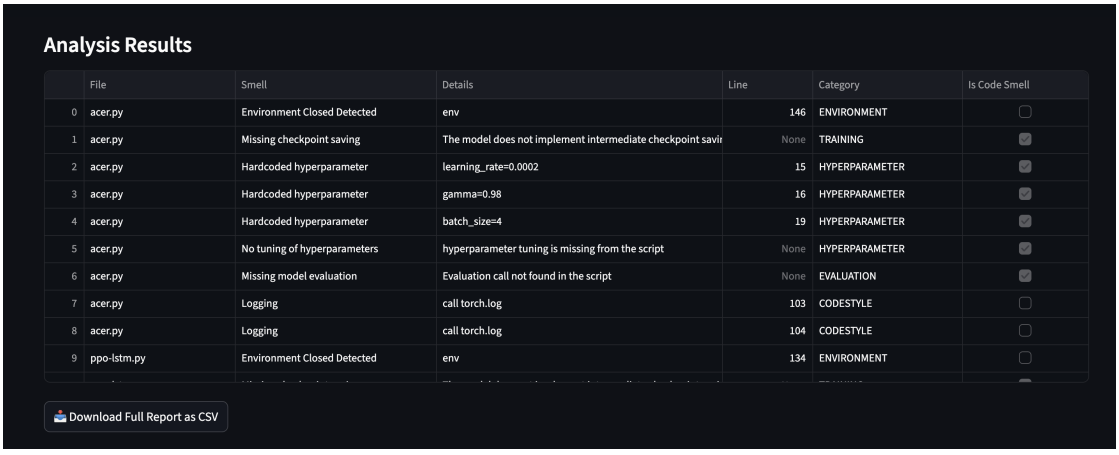

Figure 5.6.: Full report - web UI interface

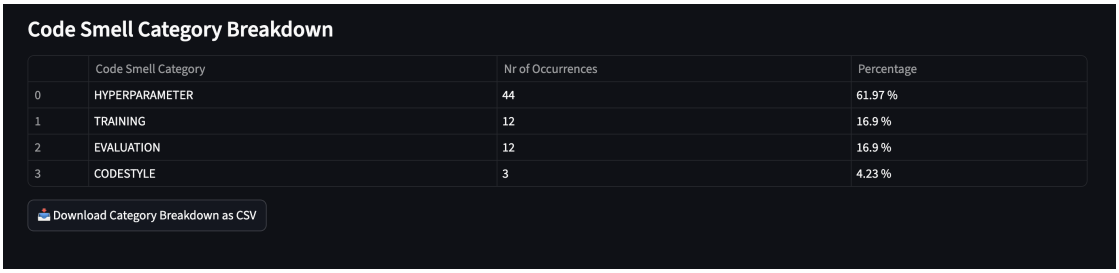

Figure 5.7.: Category overview section - web UI interface

Figure 5.8.: Distribution of smells by category - web UI interface

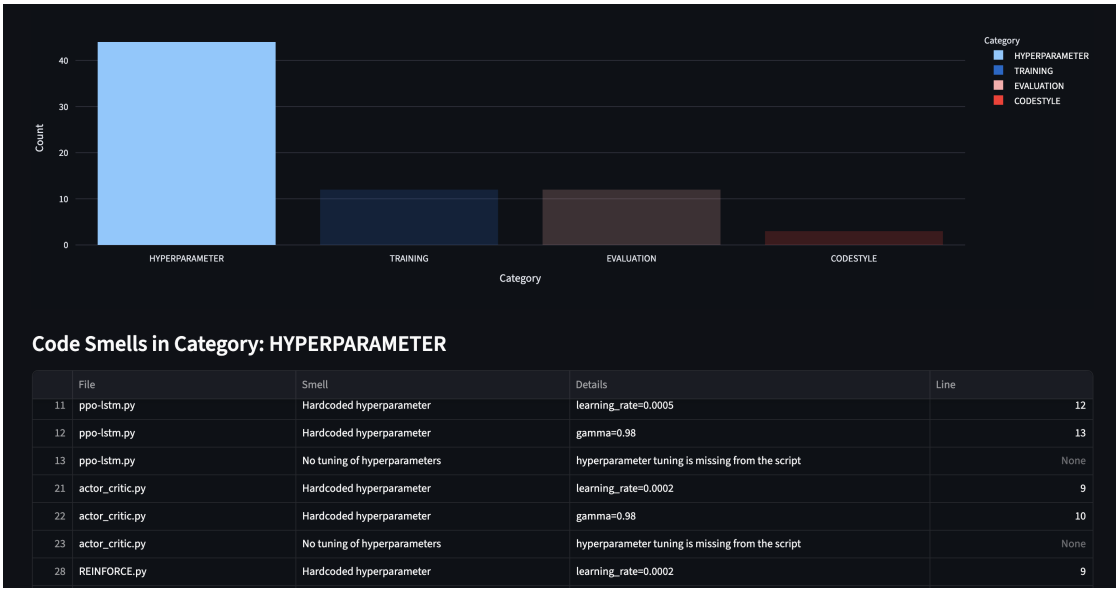

Figure 5.9.: Close up category section - web UI interface

The source code of the tool can be found at https://github.com/filipgeorgiana/rl-code-smell-detector [S20].

6. RL code smells detection tools benchmark

6.1. Evaluation of the RL Code Smell Detection Tool Performance

We applied the tool to 51 open-source projects from GitHub, which spanned different RL domains and topics. We used GitHub as a source for open-source RL project mining, by searching for repositories that include reinforcement learning tags. The projects were curated based on the same criteria we conducted the preliminary study, which is:

- Relatively recent and maintained projects: in which the last commit should not be older than 3 years.
- Reasonable number of stars on the repository: at least 50 stars on GitHub.
- Implementation of the projects in Python programming language. Python was selected because it is the prevalent programming language in RL research and industry applications, but also because of the usage of the AST Python module, therefore the tool is designed for Python based projects only.
- Use of reinforcement learning libraries and reinforcement learning as GitHub tags. We only selected projects that make use of RL libraries or frameworks.

6.2. Full Evaluation on MinimalRL

In order to assess the quality and accuracy of the RL code smell detection tool, we calculated the precision and recall [S13] for 5 repositories, first being the *minimalRL* repository [S12], since doing this process over the all the 51 repositories would require extreme manual effort. The minimalRL project [S12] was chosen for manual inspection and assessment of the results because it is a rather small and pedagogical code base that implements twelve standard reinforcement learning algorithms as standalone Python scripts. The repository consists of 12 Python files with a total size of 62 KB, which is small enough for a manual inspection while still capturing representative RL coding patterns.

The tool detected 71 code smell instances across these files prior to manual evaluation. After manual inspection, we established a ground truth of 116 RL code smells. Comparing the tool output against this ground truth resulted in 69 true positives, 2 false positives,

and 47 false negatives. This corresponds to an overall precision of 97.2%, recall of 59.5%, and an F1-score of 0.738 [S14].

File	TP	FP	FN	Precision	Recall	F1
a2c.py	4	1	5	80.0%	44.4%	0.571
a3c.py	4	1	5	80.0%	44.4%	0.571
acer.py	6	0	4	100.0%	60.0%	0.750
actor_critic.py	5	0	2	100.0%	71.4%	0.833
ddpg.py	8	0	5	100.0%	61.5%	0.762
dqn.py	7	0	1	100.0%	87.5%	0.933
ppo-continuous.py	7	0	5	100.0%	58.3%	0.737
ppo-lstm.py	5	0	5	100.0%	50.0%	0.667
ppo.py	5	0	5	100.0%	50.0%	0.667
sac.py	8	0	5	100.0%	61.5%	0.762
REINFORCE.py	5	0	1	100.0%	83.3%	0.909
vtrace.py	5	0	4	100.0%	55.6%	0.714
TOTAL	**69**	**2**	**47**	**97.2%**	**59.5%**	**0.738**

Table 6.1.: Per-file evaluation results

As shown in the above table 6.2, precision reaches 100% in 10 out of the 12 scripts. The two exceptions, `a2c.py` and `a3c.py`, each produce one false positive. The file `dqn.py` achieves the highest recall among all files (87.5%), as there is only a false negative due to a hard-coded hyperparameter value that goes undetected by the tool.

Per-Category Results

A breakdown by category reveals significant variation in detection performance.

Category	TP	FP	FN	Precision	Recall	F1
HYPERPARAMETER	44	0	38	100.0%	53.7%	0.698
TRAINING	12	0	0	100.0%	100.0%	1.000
EVALUATION	10	2	0	83.3%	100.0%	0.909
CODESTYLE	3	0	9	100.0%	25.0%	0.400

Table 6.2.: Per-category evaluation results

The TRAINING category achieves perfect performance, while EVALUATION also performs strongly despite two false positives. In contrast, HYPERPARAMETER detection exhibits high precision but moderate recall, and CODESTYLE shows significantly lower recall due to limitations in the logging detection strategy, caused by falsely labeling certain logarithm functions as logging patterns.

6.3. Full Evaluation on PPO-PyTorch

In order to assess whether the results obtained on minimalRL [S12] generalize well on other structural repositories, we performed another analysis on a different repository: PPO-PyTorch [S26].

The PPO-Pytorch[S26] project is a Proximal Policy Optimisation (PPO) [28] algorithm implementation with support for both discrete and continuous action spaces. Unlike the minimalRL [S12] project, in which every file is contains a training part, PPO-PyTorch[S26] is organized in a more modular approach, containing scripts dedicated to training, evaluation and algorithm implementation.

Our RL code smell detector tool flagged five Python scripts for evaluation, two of which no code smells were detected and three files were classified as containing RL code smells, and after the manual inspection of the repository we constructed a ground truth of 32 code smells. The tool raised 20 detections, of which 17 were confirmed as true positives and 3 as false positives, and 15 false negatives. The overall precision is 85%, recall 53.1%, and F1-score 0.654.

File	TP	FP	FN	Precision	Recall	F1
`train.py`	6	0	9	100.0%	40.0%	0.571
`test.py`	5	2	3	71.4%	62.5%	0.667
`make_gif.py`	6	1	3	85.7%	66.7%	0.750
`PPO.py`	0	0	0	—	—	—
`plot_graph.py`	0	0	0	—	—	—
Total	17	3	15	85.0%	53.1%	0.654

Table 6.3.: Per-file precision, recall, and F1 on PPO-PyTorch.

On the train.py we achieve a precision of 100%, which means that every detection in the main training script corresponds to a RL code smell that our detector correctly flags. However, precision is lower in the other scripts because the tool flags some issues in non-training and evaluation-related files that do not represent genuine code smells in context, the tool wrongfully flags code smells such as missing checkpoint saving and missing model evaluation. The recall of 40% on train.py is explained by the same vocabulary limitation in detected hard-coded hyperparameters as discussed in the minimalRL [S12] evaluation, our RL detector tool correctly detects hard-coded hyperparameters which are part of its known vocabulary but, in case of hard-coded hyperparameter names outside the vocabulary, those go undetected.

Category	TP	FP	FN	Precision	Recall	F1
HYPERPARAMETER	12	0	15	100.0%	44.4%	0.615
EVALUATION	2	1	0	66.7%	100.0%	0.800
TRAINING	0	2	0	0.0%	—	0.000
CODESTYLE	3	0	0	100.0%	100.0%	1.000

Table 6.4.: Per-category precision, recall, and F1 on PPO-PyTorch.

Looking at per category results, the CODESTYLE category achieves a perfect F1 score. The detector tool correctly identified improper logging strategies, because the script uses `print()` statements for logging messages or track progress. Regarding the HYPERPARAMETER category, everything we detect as hyperparameter smells is a true positive, however, the recall of 44.4% is due to the missing hyperparameter terms in the tool's vocabulary. For the EVALUATION category, precision is 66.7% with recall of 100%. The single false positive is `test.py`: the tool flagged it for "missing model evaluation" despite the fact that `test.py` is the evaluation module, the `test()` function. The tool did not recognize it as evaluation because it did not parse function definitions. Regarding the two false positives in the TRAINING category, they appear in the `test.py` and `make_gif.py`, both files are flagged for missing checkpoint saving, which is true and consistent from the RL code tool's perspective implementation, because the tool does not distinguish between training scripts and non-training scripts. We classify this detection as a false positive because the ground truth asks whether the smell carries genuine engineering significance, thus this is subjective and can impose a threat to validity.

6.4. Full Evaluation on Papers-in-100-Lines-of-Code

In order to strengthen the reliability of the RL code smell detector tool and test it further, we conducted a third evaluation on the Papers-in-100-Lines-of-Code repository [S27], which is an active maintained project containing RL algorithm implementations, each designed to reproduce a published paper in approximately 100 lines of Python code. The tool flagged four scripts as containing code smells. All four files are standalone RL training scripts, placing this repository in the same structural category as minimalRL[S12].

File	TP	FP	FN	Precision	Recall	F1
`ddqn.py`	3	1	0	75.0%	100.0%	0.857
`ppo.py`	3	0	0	100.0%	100.0%	1.000
`dqn.py` (HLC paper)	3	1	0	75.0%	100.0%	0.857
`dqn.py` (Playing Atari)	4	1	0	80.0%	100.0%	0.889
Total	13	3	0	81.3%	100.0%	0.897

Table 6.5.: Per-file precision, recall, and F1 on Papers-in-100-Lines-of-Code.

Category	TP	FP	FN	Precision	Recall	F1
Hyperparameter	4	0	0	100.0%	100.0%	1.000
Evaluation	4	0	0	100.0%	100.0%	1.000
Codestyle	4	0	0	100.0%	100.0%	1.000
Training	1	0	0	100.0%	100.0%	1.000
Agent	0	3	0	0.0%	—	0.000
Environment		—		—	—	—

Table 6.6.: Per-category precision, recall, and F1 on Papers-in-100-Lines-of-Code.

This repository is the first, in the tool's evaluation process, to trigger AGENT category detections, which we classified as false positives because our tool flags `env.action_space.sample()` as "random behavior detected for sampling", which is a RL specific code smell designed to identify agents that rely exclusively on random action selection rather than a learned policy, but in all three cases, the call appears inside an epsilon-greedy exploration conditional, which is a strategy in RL to mitigate the exploration vs. exploitation trade-off [S30].

For the four categories HYPERPARAMETER, EVALUATION, TRAINING, and CODESTYLE, the precision on Papers-in-100-Lines-of-Code [S27] is 100%, consistent with the pattern that the tool performs reliably on standalone RL training scripts.

6.5. Full Evaluation on Reinforcement-learning-an-introduction repository

A fourth evaluation was conducted on the reinforcement-learning-an-introduction repository [S28], a collection of Python implementations accompanying Sutton and Barto's textbook Reinforcement Learning: An Introduction (2nd ed.) [32].

Our RL tool raised 14 positive detections, across two files, from which, 11 were confirmed as true positives and 3 as false positives, leading to a precision of 78.6%. A further 7 smell instances present in the code were not detected by the tool, giving a recall of 61.1% and an F1-score of 0.688.

File	TP	FP	FN	Precision	Recall	F1
`short_corridor.py`	7	3	1	70.0%	87.5%	0.778
`lambda_effect.py`	4	0	6	100.0%	40.0%	0.571
Total	11	3	7	78.6%	61.1%	0.688

Table 6.7.: Per-file precision, recall, and F1 on reinforcement-learning-an-introduction.

The TRAINING, EVALUATION, and CODESTYLE categories all achieve perfect precision and recall: both files lack checkpoint saving, both miss the evaluation step,

Category	TP	FP	FN	Precision	Recall	F1
Hyperparameter	5	3	7	62.5%	41.7%	0.500
Training	2	0	0	100.0%	100.0%	1.000
Evaluation	2	0	0	100.0%	100.0%	1.000
Codestyle	2	0	0	100.0%	100.0%	1.000

Table 6.8.: Per-category precision, recall, and F1 on reinforcement-learning-an-introduction.

and both lack any form of structured logging. All six detections in these categories are confirmed true positives with no missed instances.

The HYPERPARAMETER category is the only source of error, and it contributes all three false positives and all seven false negatives, all caused by the tool misclassifying hard-coded hyperparameters, or completely missing hyperparameters that are hard-coded inside the constructors body, assignment like discount factor and exploration rate.

6.6. Full Evaluation on ICCV2019-LearningToPaint

For our fifth evaluation, we chose a repository also on the smaller side, but from a different architecture perspective than the others. ICCV2019-LearningToPaint [S29] has a modular architecture and is a RL application design to make an agent paint like a human. Our tool analyzed two nearly identical training entry points (`baseline/train.py` and `baseline_modelfree/train.py`), achieving 66.7% precision, 100% recall, and F1=0.800 across both files.

File	TP	FP	FN	Precision	Recall	F1
`baseline/train.py`	4	2	0	66.7%	100.0%	0.800
`baseline_modelfree/train.py`	4	2	0	66.7%	100.0%	0.800
Total	8	4	0	66.7%	100.0%	0.800

Table 6.9.: Per-file precision, recall, and F1.

Category	TP	FP	FN	Precision	Recall	F1
Training	4	2	0	66.7%	100.0%	0.800
Hyperparameter	2	0	0	100.0%	100.0%	1.000
Evaluation	0	2	0	0.0%	—	—
Codestyle	2	0	0	100.0%	100.0%	1.000

Table 6.10.: Per-category precision, recall, and F1.

The two false positives per file share the same root cause, which is cross-file blindness, since our tool evaluates `train.py` in isolation without considering other modules. For example, checkpoint saving is implemented in `ddpg.py` (`save_model()` with `torch.save()`) and called from `train.py`, but the tool does not recognize this, since it does a per-file analysis. Similarly, the evaluation phase is implemented in a dedicated `Evaluator` class and invoked from `train.py`, but our tool also flags the evaluation as missing. Both false positives would disappear if the tool considered the full module graph rather than individual files, which currently does not happen, since the tool only considers per-file structure.

The eight true positives are correctly identified: no tuning of hyperparameters, no structured Python logging, and training-evaluation coupling where `train()` directly calls the evaluator and triggers model saving inside the training loop.

6.7. Overall Reliability

The evaluation results indicate that RL code smell detector tool achieves overall a good precision, which means that detected RL smells are highly reliable, and the tool rarely flags an issue that is not a RL code smell. The precision drop on PPO-PyTorch [S26] and ICCV2019-LearningToPaint [S29] is explained entirely by false positives due to file-type blindness, and not by any deterioration in the tool's core detection logic, which is a takeaway point for future work, improvements and refinements on the tool's detection logic.

Another takeaway from the evaluation results is the limited vocabulary hyperparameter detector which periodically misses algorithm specific parameter names. We notice the tool behaves extremely well on projects that contain full RL scripts, in case of a modular architecture and project, the precision of our tool drops significantly because it is not modular-aware of the whole context of the application.

Repository	**Precision**	**Recall**	**F1**
minimalRL	97.2%	59.5%	0.738
PPO-PyTorch	85.0%	53.1%	0.654
Papers-in-100-Lines-of-Code	81.3%	100.0%	0.897
reinforcement-learning-an-introduction	78.6%	61.1%	0.688
LearningToPaint	66.7%	100.0%	0.800
Mean	81.8%	74.7%	0.755

Table 6.11.: Overall precision, recall, and F1 across evaluated repositories.

The false negatives identified in the HYPERPARAMETER category are of type hard coded hyperparameter and are due to the current limited capacity of the detector. The vocabulary, of hard-coded hyperparameters possible names, is manually maintained and more standardized, thus using variable names less standard can lead to some hard-coded

hyperparameters not being successfully identified by the detection tool. This was an issue constantly observed in the evaluated repositories.

The issue identified in CODESTYLE category suffers from a flawed heuristic that incorrectly interprets mathematical logarithm functions (e.g., `torch.log`) as evidence of logging. This leads to systematic falsely detected logging practices, which is more visible in the minimalRL[S12] project.

Furthermore, another very important thing to consider is cross-file blindness: in modular projects like ICCV2019-LearningToPaint [S29] and PPO-PyTorch [S26], genuine smells are absent from individual files because responsibilities are distributed across modules, which our tool cannot track well.

All in all, the results suggest that our RL code smell detector tool performs reliably on standalone RL training scripts. As of now, certain improvements can be made, like including RL file type awareness, extending the hyperparameter vocabulary, improving the logging and agent smell detectors.

7. Analysis of RL Code Smell Detection Results

In the following chapter, we discuss the results obtained from the RL code smell detection tool and conduct an empirical analysis. The tool was tested and run across 51 open-source repositories. The repositories were curated based on the same criteria described in the section 4.4.1. The objective of the analysis is to study correlations between the smells and their repository-level characteristics, ultimately identifying structural patterns, and connections between certain RL specific code smells.

In total, four types of analysis were performed on the results dataset, focusing on:

- the distribution of code smells across predefined categories
- their relationship with repository attributes such as age, size, popularity, and contributor count
- correlations between code smell categories
- variations across application domains.

Note: for performing the analysis step and process, we first gather all the resulted reports of the analyzed repositories, both full report and per-category report, then centralized the data and computed python scripts with the help of an AI Code Agent in order to facilitate the calculation or correlations and centralization of data.

7.1. Overview of Code Smell Distribution

In the analysis of the 51 selected repositories, a total of 4,872 code smell occurrences and 6,392 heuristics detections were identified, distributed across the six predefined categories: HYPERPARAMETER, EVALUATION, TRAINING, CODESTYLE, AGENT and ENVIRONMENT.

Four categories account for approximately 97.5% of all detected smells and appear in nearly all repositories:

- HYPERPARAMETER: 29.95%
- EVALUATION: 24.98%
- TRAINING: 22.66%

- CODESTYLE: 19.87%

Their consistent presence across repositories suggests that these smells are not isolated issues, but rather structural characteristics in RL codebases.

In contrast, the AGENT (2.50%) and ENVIRONMENT (0.04%) categories are significantly less frequent and appear in only a subset of repositories, suggesting that these smells are more context-dependent and likely tied to specific architectural or domain-related design choices.

Category	**Occurrences**	**Percentage**
HYPERPARAMETER	1,459	29.95%
EVALUATION	1,217	24.98%
TRAINING	1,104	22.66%
CODESTYLE	968	19.87%
AGENT	122	2.50%
ENVIRONMENT	2	0.04%
Total	**4,872**	**100%**

Table 7.1.: Distribution of RL code smells across categories in 51 repositories.

To examine where these categories occur most frequently, we analyzed the distribution of detected smells across different file types. The results show that most smell categories are concentrated in algorithm implementation files, environment files, and test files, while fewer smells occur in model, evaluation, and utility/helper files. AGENT smells differ from this general pattern, as they occur most frequently in environment files and test files.

File Type	**Hyperparameter**	**Training**	**Evaluation**	**Codestyle**	**Agent**
Algorithm	47.6%	43.7%	45.7%	45.6%	18.9%
Environment files	20.6%	24.8%	23.2%	24.7%	46.7%
Test files	14.0%	16.5%	15.3%	18.0%	32.0%
Training scripts	8.2%	7.7%	8.8%	3.9%	0.8%
Config / args files	4.0%	1.4%	1.2%	1.5%	1.6%
Model / network	2.2%	2.2%	2.2%	2.4%	0.0%
Evaluation files	2.1%	2.4%	2.3%	2.6%	0.0%
Utility / helper	1.4%	1.4%	1.3%	1.3%	0.0%

Table 7.2.: Distribution of detected smells by file type and category (%).

7.2. Relationship Between Repository Age and Code Smells

To investigate whether code smells accumulate over time, as a repositories ages, we analyzed the correlation between repository age and total smell count. The results indicate a weak negative correlation (Pearson $r = -0.237$), which is not statistically significant.

Contrary to expectations that older repositories are more "smelly" than relatively new projects, older repositories do not explicitly have higher number of code smells. In fact, some of the oldest repositories contain relatively few smells, but we can also consider the fact that those might be either not currently popular repositories or are less complex, while several younger repositories have high smell counts. This suggests that code smell prevalence in RL systems is not primarily driven by the age of a project.

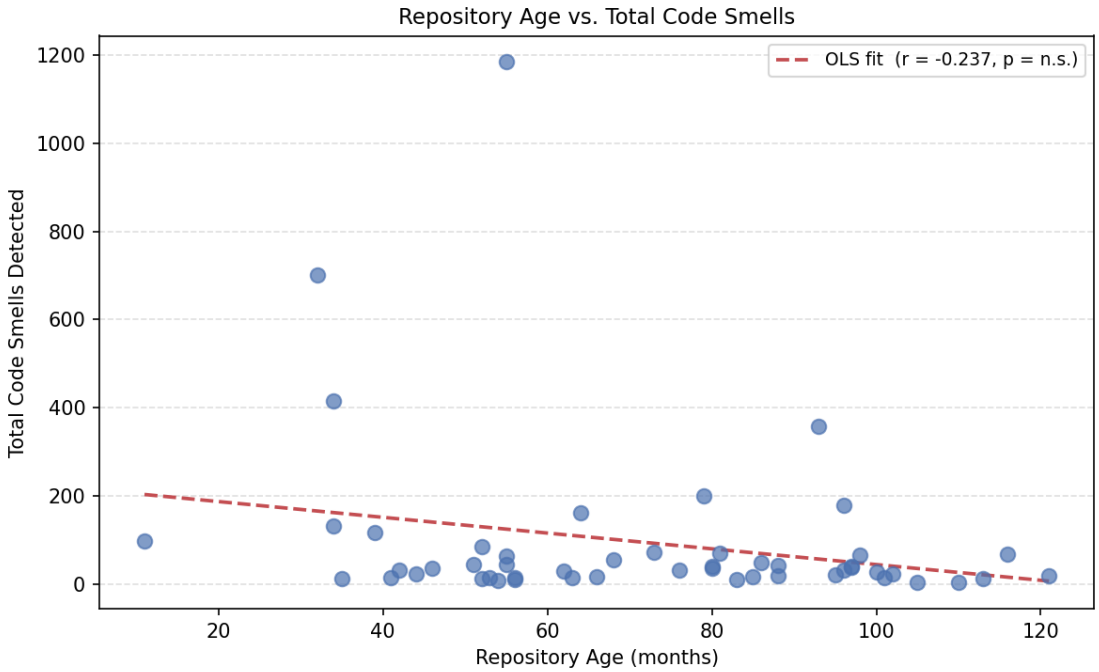

Figure 7.1.: Repository age versus total code smells detected

7.3. Impact of Repository Characteristics

We also analyzed and evaluated if there is any correlation between identified RL code smells (from a category perspective) and certain attributes of the repositories, such as: number of stars, number of contributors, size. The results of the analysis show that the code base size is the only significant predictor of code smells occurring across all defined categories.

A strong correlation was observed between repository size and multiple categories, particularly TRAINING ($r = 0.645$), CODESTYLE ($r = 0.466$), and overall smell count ($r = 0.429$).

In contrast, repository popularity (measured by GitHub stars) shows negligible correlation with code smells and the number of contributors shows weak, non-significant correlations. These findings suggest that RL code smells are more prevalent in larger projects, while repository popularity, and number of contributors do not significantly predict the presence of RL code smells across categories.

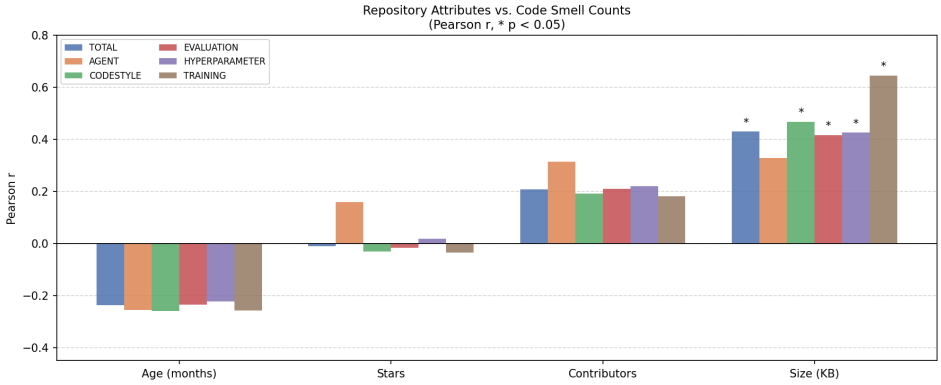

Figure 7.2.: Pearson correlation between repository attributes and code smell counts, by category (n = 51).

Repository size is the only repository attribute that shows statistically significant correlations with the number of code smells present, across all categories ($p < 0.05$), with the strongest effect observed for TRAINING smells ($r = 0.645$). Repository age shows weak negative correlations across categories (approximately r = -0.22 to -0.26), but these correlations are not statistically significant. Stars show negligible and non-significant correlations with all smell counts ($|r| < 0.04$), indicating that repository popularity is not associated with code quality as measured by the detector. Contributors show weak positive, non-significant correlations. Overall, the results indicate that smell accumulation is associated primarily with codebase size rather than repository maturity, visibility, or team size.

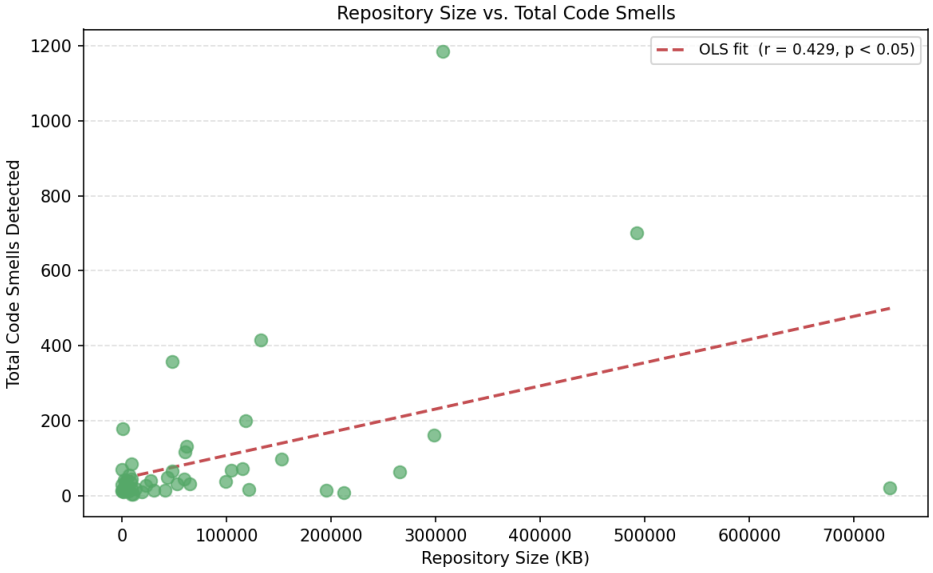

Figure 7.3.: Repository size versus total number of detected code smells.

7.4. Correlation Between Code Smell Categories

We also examined inter-category correlations to determine whether certain smell categories tend to co-exists. Specifically, we investigated whether repositories exhibiting a high number of smells in one category are also likely to exhibit elevated counts in another.

To this end, we performed a pairwise correlation analysis across all six smell categories using two complementary measures: the count-based Pearson correlation [29] and the presence-based Phi coefficient.

The Pearson correlation operates on raw detection counts, measuring whether the volume of smells in one category covaries with the volume in another category of code smells, across repositories. The Phi coefficient, in contrast, operates on binary indicators: each repository's count is reduced to 1 if a category is detected at all, and 0 if it is absent. Phi therefore captures whether two categories tend to appear together across repositories, independently of how many instances each contains, making it a measure of co-occurrence. The analysis revealed that there is a strong relationship between the four prevalent categories (HYPERPARAMETER, EVALUATION, TRAINING, and CODESTYLE).

The count-based correlation analysis reveals strong positive relationships among the four core smell categories: EVALUATION, TRAINING, HYPERPARAMETER, and CODESTYLE. All pairwise correlations are statistically significant ($p < 0.05$), with Pearson correlation coefficients ranging from $r = 0.965$ to $r = 0.995$.

These results indicate that repositories exhibiting a high number of smells in one core category tend to show similarly elevated counts across the others. This pattern indicates that these categories do not occur independently, but rather covary as part of a shared underlying factor in how the repository was developed.

Category Pair	**Pearson** r
EVALUATION × TRAINING	0.995
EVALUATION × HYPERPARAMETER	0.987
CODESTYLE × TRAINING	0.983
Range (all pairs)	0.965 – 0.995

Table 7.3.: Strongest correlations between core smell categories (count-based)

The strongest observed relationship is between EVALUATION and TRAINING ($r = 0.995$), indicating an almost perfect linear association. Similarly, high correlations are observed for EVALUATION with HYPERPARAMETER ($r = 0.987$) and for CODESTYLE with TRAINING ($r = 0.983$).

The consistently high correlation values indicate that these four categories are tightly coupled rather than independent sources of code quality issues. Rather than occurring in isolation, the categories appear to capture related aspects of the same underlying implementation structure in RL codebases, which suggests that these smell categories are closely linked to the design and implementation of RL training pipelines.

AGENT and ENVIRONMENT smells behave differently from the other categories. AGENT smells are more strongly influenced by repository size than by systematic co-occurrence with other smell categories. ENVIRONMENT smells show no meaningful correlation with any category, primarily because they occur very rarely in the dataset.

In figures 7.4 and 7.5 we illustrate the Pearson correlation matrix based on raw smell counts and the Phi coefficient matrix based on binary smell-presence indicators. The four core categories, EVALUATION, TRAINING, HYPERPARAMETER, and CODESTYLE, form a strongly correlated cluster in the Pearson matrix (r = 0.965–0.995, all $p < 0.05$), reflecting a shared repository-size effect. ENVIRONMENT is uncorrelated with all other categories (r approximately -0.06 to 0.003). In contrast, the Phi coefficient matrix provides limited additional insight because the core smell categories are present in almost every repository. In particular, EVALUATION and HYPERPARAMETER appear in all 51 repositories, while TRAINING appears in 49 out of 51 repositories. As a result, binary smell presence has little discriminating power, making the presence-based analysis less informative than the count-based Pearson analysis.

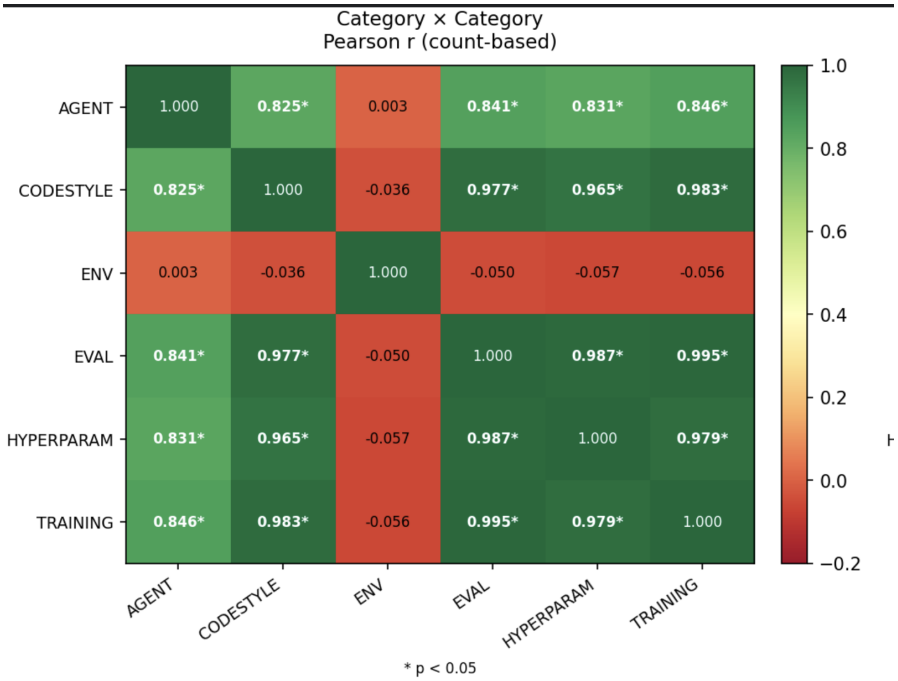

Figure 7.4.: Category level correlation Pearson coefficient

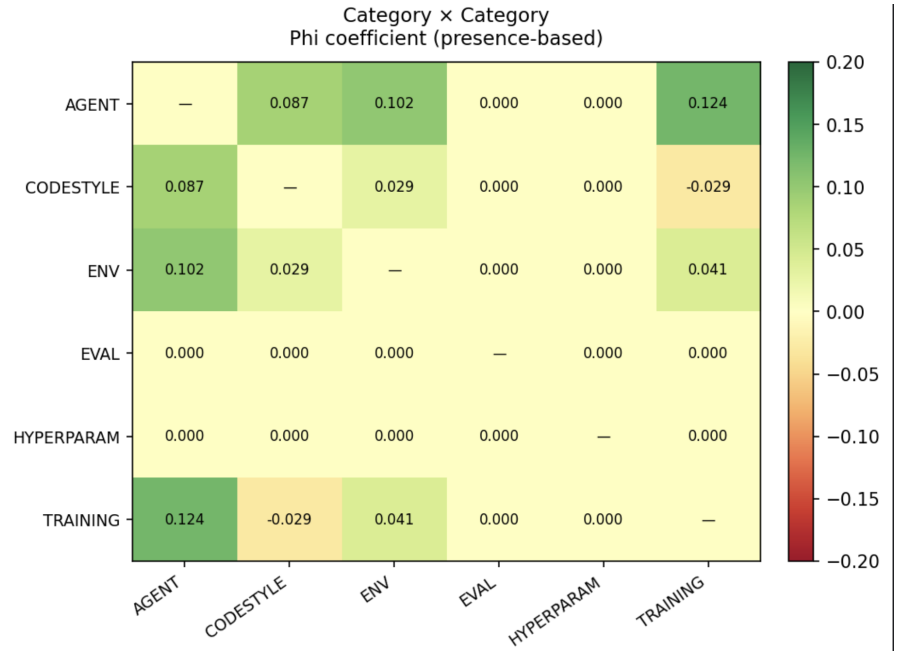

Figure 7.5.: Category level correlation Phi coefficient

7.5. Topic-Based Analysis

For the domain-based analysis, we used the Github tags attached to each repository metadata to automatically determine the domain, and then grouped the repositories based on the tags. The repositories are spread across the following different domains, topics:

- Game playing
- Robotics
- Autonomous Driving
- NLP / LLM
- Computer Vision
- Operations / Finance

One thing to note here is that certain categories, described by tags (such as "Distributed", "Large", "Framework"...) were excluded from the classification as they rather describe implementation characteristics and not domains. In case a repository was linked to two different domains, we assigned it to the one fits better from a semantical point of view. The repositories whose tags did not match any application domain, or were missing topic tags, were left unclassified and excluded from this analysis, this left us with 25 repositories, which belonged to the 6 domains listed above. Therefore, this poses a threat to validity since the data set for this analysis is rather limited and the results might not be able to generalize statistically.

The table 7.5 shows the mean proportion of each smell category per domain. From the results, we can observe the most visible pattern is an inversely proportional relationship between HYPERPARAMETER smells and TRAINING smells across domains. Repositories belonging to the Computer Vision domain have the highest average HYPERPARAMETER proportion at 40.0%, and the lowest TRAINING proportion at 15.2%. Game Playing repositories show the opposite: HYPERPARAMETER accounts for only 26.9% of smells on average, while TRAINING reaches 26.2%, the highest of any domain. Robotics/Control follows a similar pattern to Computer Vision, with HYPERPARAMETER at 35.7% and TRAINING at 18.5%. This suggests that repositories where the training loop itself is central, such as game-playing agents, tend to accumulate more training-related issues, while repositories that adapt existing architectures from other fields tend to carry more hard-coded parameter debt instead.

EVALUATION and CODESTYLE smells show little variation across domains, suggesting that RL code smells belonging to these categories are not domain-specific problems and their appearance is not a consequence of the application area. AGENT and ENVIRONMENT smells are marginal across all domains, with AGENT reaching at most 5.8% in Computer Vision and ENVIRONMENT never exceeding 0.6%.

Domain	N	HP.	EVAL.	TRAIN.	CS	AGENT	ENVIRON.
Computer Vision	4	40.0%	19.9%	15.2%	19.1%	5.8%	0.0%
Robotics / Control	5	35.7%	27.4%	18.5%	17.8%	0.0%	0.6%
Autonomous Driving	4	27.0%	24.2%	23.1%	23.0%	2.3%	0.4%
Game Playing	9	26.9%	26.4%	26.2%	19.6%	1.0%	0.0%
NLP / LLM	2	26.8%	26.8%	23.2%	23.2%	0.0%	0.0%
Operations / Finance	1	25.7%	25.7%	25.7%	23.0%	0.0%	0.0%

N: Number of repositories
HP: HYPERPARAMETER category
CS: CODESTYLE category

Table 7.4.: Mean proportion of each smell category per domain.

7.6. Conclusions

7.6.1. Summary of findings

To summarize the findings, we have found several key insights:

1. The size of the codebase is the predominant factor that influences code smell counts.
2. The four main categories (HYPERPARAMETER, EVALUATION, TRAINING, CODESTYLE) form a tightly coupled structural group.
3. EVALUATION and HYPERPARAMETER smells have a very high presence rate.
4. HYPERPARAMETER smells are the most prevalent.
5. AGENT smells are primarily associated with large and complex architectures.
6. ENVIRONMENT smells are rare and require further investigation.
7. The application domain appears to influence the relative distribution of smell categories, although the limited sample size reduces generalizability.
8. Repository popularity and age are not reliable indicators of code quality in RL systems.

Overall, this empirical analysis demonstrates that reinforcement learning code smells are not random defects but result from strong structural patterns. The findings highlight the importance of considering codebase size and architectural characteristics when analyzing code quality in RL systems. Furthermore, the consistent presence of certain smell categories suggests opportunities for targeted improvements in RL development practices, particularly in hyperparameter management, evaluation procedures, and training pipeline design.

8. Threats to Validity

8.1. Threats to the Empirical Analysis

This study has several limitations that should be acknowledged.

- **Conclusion validity.** The empirical analysis is based on 51 repositories. While this number is sufficient for an exploratory study, it limits the statistical power of the correlation and distribution analyses. Therefore, the observed relationships between repository characteristics and code smell prevalence should be interpreted cautiously.
- **Construct validity.** The domain analysis relies on repository metadata and GitHub topics to determine project domains. Since these tags are provided by repository maintainers and are not always complete or consistent, the extracted domains may not fully reflect the actual purpose or application area of each project.
- **External validity.** Some repositories could not be clearly assigned to a domain due to insufficient metadata. As a result, the domain analysis was conducted on a subset of 25 projects. Therefore, the topic-based findings should be viewed as an exploratory description of the available data rather than a generalizable domain study.
- **Construct validity.** All detected code smells are treated equally in the quantitative analysis. The analysis does not account for differences in severity, impact, or practical relevance. Consequently, a minor smell and a smell with substantial effects on reproducibility or training validity contribute equally to the reported counts.
- **Construct validity.** Certain smell categories, particularly ENVIRONMENT and AGENT, have limited representation in the dataset. This restricts the strength of conclusions that can be drawn about these categories compared with more frequently occurring categories such as HYPERPARAMETER, EVALUATION or TRAINING.

8.2. Threats to the Tool Evaluation

- **Construct validity.** The ground truth for the evaluated repositories was constructed by a single annotator. Since the identification of RL specific code smells involves judgment, especially in context-dependent cases, this introduces a degree of subjectivity. This threat could be reduced in future work by involving a second annotator.

- **External validity.** Although the tool was evaluated on multiple repositories, the evaluation still covers only a limited subset of the analyzed RL projects. The selected repositories represent different structural characteristics, including standalone scripts and modular projects, but they do not cover the full diversity of larger framework-based or domain-specific RL systems. Therefore, the observed precision and recall may not fully generalize to all RL codebases.

- **Construct validity.** Classifying certain detections as false positives requires contextual judgment. For example, in modular repositories, a file may resemble a training script even though it is used only for evaluation, visualization, or auxiliary functionality. In such cases, the heuristic may be technically correct at the file level but not meaningful from an engineering perspective.

- **Internal validity.** Some false positives and false negatives are caused by limitations of the current static analysis implementation. For example, the tool has limited file-type awareness, limited cross-file analysis, and relies on fixed vocabularies for certain smell detectors. These implementation choices influence the measured accuracy of the tool.

- **Construct validity.** The implemented smell detectors cover only a subset of possible RL specific code smells. Each category could be extended with additional smells, and some context-dependent smells may require more advanced analysis techniques. Therefore, the reported results should be interpreted as a lower-bound approximation of smell prevalence rather than a complete representation of all possible RL code quality issues.

9. Conclusions and Future Work

9.1. Conclusions

This final chapter summarizes the findings of the thesis by revisiting the initial research questions and discussing the results obtained in this study. At the beginning of the thesis, we proposed three research questions. The first research question was:

> **RQ1: What types of code smells are most prevalent in reinforcement learning systems?**

To answer this question, we conducted an empirical analysis showing that the most prevalent reinforcement learning code smells are related to hyperparameters, particularly hard-coded hyperparameter values and missing hyperparameter tuning. In addition, smells related to evaluation, training, and code style also represent a significant proportion of the detected smells. Together, these four categories account for approximately 97.5% of all detected code smells across the 51 analyzed repositories. Overall, we detected 4,872 RL-specific code smells. The results suggest that the analyzed RL repositories rely heavily on manually configured experimental parameters, often without sufficient abstraction or configuration management.

Moving on to the second research question which focussed on:

> **RQ2: How can we categorize the different code smells found in RL systems?**

To address this question, this thesis proposed a categorization consisting of six categories: HYPERPARAMETER, EVALUATION, TRAINING, CODESTYLE, AGENT, and ENVIRONMENT. These categories were designed to capture recurring software quality issues specific to RL systems while remaining broad enough to apply across different repositories and domains. The results show that this categorization provides a useful structure for analyzing RL-specific code smells. The four prevalent categories were found consistently across repositories of different sizes and purposes, suggesting that they represent common structural characteristics of RL codebases rather than isolated implementation issues. The less frequent AGENT and ENVIRONMENT categories capture smells that are more context-dependent and tied to specific architectural choices. Overall, the proposed categories align well with the results of the empirical analysis.

And lastly, the third research question:

> **RQ3: How does the prevalence of code smells correlate with project characteristics such as size, contributor count, age, and other factors?**

The correlation analysis revealed strong relationships between smell categories, particularly among the four main categories. Repositories with high counts of one smell type also tended to show high counts in the other main categories. These findings suggest that code smells in RL systems do not occur independently, but often appear together as part of broader structural quality issues. Repository size showed the strongest relationship with overall smell count, indicating that larger repositories generally contain more detected smells, which is expected given their increased code volume and complexity. Contributor count showed weak positive correlations with smell prevalence, although these correlations were not statistically significant. Repository age showed weak negative correlations, but these were also not statistically significant. In addition to the empirical study, this thesis evaluated the reliability of the proposed detection tool on five repositories. Across these repositories, the tool achieved a mean precision of 81.7%, a mean recall of 75.4%, and a mean F1-score of 0.761. These results suggest that the tool can reliably identify many RL specific code smell instances, especially in standalone training scripts. However, the evaluation also showed important limitations, particularly vocabulary limitations in the hyperparameter detector and reduced precision in modular projects due to file-type and cross-file analysis limitations.

Overall, this thesis provides an empirical investigation of code smells in reinforcement learning systems and proposes a static analysis tool for detecting recurring RL implementation issues. The results show that most detected smells are related to hyperparameters, evaluation, training, and code style. These findings suggest that many RL projects would benefit from improved configuration management, clearer evaluation procedures, more systematic experiment management, and stronger engineering practices. The proposed categorization and detection tool provide a foundation for further research on software quality in RL systems and for future tools that support maintainable, reproducible, and reliable RL development.

9.2. Future Work

Regarding future work, several directions for improving and extending the tool can be considered in order to gain further insights into the quality of RL systems.

One of the clearest limitations identified during the evaluation of the code smells detector tool is the hyperparameter detector. The current implementation relies heavily on matching standard variable names, which means it misses many valid hyperparameters when different naming conventions are used. Extending the vocabulary would improve recall immediately.

Another important direction for future work is the expansion of the covered code smell catalogue. Future work could extend the detection tool to cover additional types of RL code smells based on the analysis of more repositories. In this thesis, the TRAINING category mainly focuses on missing checkpoint saving, but other training-related issues may also be relevant in RL systems, such as missing gradient clipping or missing learning rate scheduling. Including these smells would provide a broader view of software quality in RL projects.

Another limitation is that the tool evaluation was conducted on a limited number of repositories, most of which are relatively small compared with large framework-based RL systems. Larger repositories often organize code differently, using configuration files, class-based architectures, cross-file dependencies, or command-line argument parsing. Future work could therefore include a broader evaluation on larger and more complex repositories to better understand how well the tool generalizes.

Another possible direction for future work would be experimenting with large language models for code smell detection. The current tool is completely rule-based and depends on predefined patterns and heuristics, which means it can miss smells that do not follow those exact patterns. LLMs could potentially help identify more context-based or semantic issues that are harder to capture with static rules alone. It would therefore be interesting to compare an LLM-based approach with the current AST-based implementation and see whether it can improve detection performance, especially for the categories with lower recall.

Finally, future versions of the tool could introduce a notion of smell severity. At the moment, all detected smells are treated equally, even though some may have a much greater impact on maintainability, reproducibility, or training stability than others. A severity ranking system could help prioritize the most important issues and make the results more useful in practice.

Bibliography

[1] Markov decision process (mdp) in reinforcement learning, 2025.

[2] Nicolás Cardozo, Ivana Dusparic, and Christian Cabrera. Prevalence of code smells in reinforcement learning projects. In *2023 IEEE/ACM 2nd International Conference on AI Engineering – Software Engineering for AI (CAIN)*, pages 37–42, 2023.

[3] 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, 2016.

[4] Peter Dayan and Bernard W Balleine. Reward, motivation, and reinforcement learning. *Neuron*, 36(2):285–298, 2002.

[5] Zihan Ding, Yanhua Huang, Hang Yuan, and Hao Dong. Introduction to reinforcement learning. In *Deep reinforcement learning: fundamentals, research and applications*, pages 47–123. Springer, 2020.

[6] 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.

[7] Farama Foundation. Gymnasium documentation – env api. `https://gymnasium.farama.org/api/env/`, 2024. Accessed: 2026-05-12.

[8] Martin Fowler. Code smell, 2006.

[9] Martin Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.

[10] Matthias Dehmer Frank Emmert-Streib. Taxonomy of machine learning paradigms: A data-centric perspective, 2022.

[11] Peter I Frazier. A tutorial on bayesian optimization. *arXiv preprint arXiv:1807.02811*, 2018.

[12] Majid Ghasemi, Amir Hossein Moosavi, Ibrahim Sorkhoh, Anjali Agrawal, Fadi Alzhouri, and Dariush Ebrahimi. An introduction to reinforcement learning: Fundamental concepts and practical applications. *arXiv preprint arXiv:2408.07712*, 2024.

[13] Imran Ghory. Reinforcement learning in board games. *Department of Computer Science, University of Bristol, Tech. Rep*, 105, 2004.

[14] Hadhemi Jebnoun, Houssem Ben Braiek, Mohammad Masudur Rahman, and Foutse Khomh. The scent of deep learning code: An empirical study. In *Proceedings of the 17th International Conference on Mining Software Repositories*, pages 420–430, 2020.

[15] Hadi S Jomaa, Josif Grabocka, and Lars Schmidt-Thieme. Hyp-rl: Hyperparameter optimization by reinforcement learning. *arXiv preprint arXiv:1906.11527*, 2019.

[16] Foutse Khomh, Massimiliano Di Penta, and Yann-Gael Gueheneuc. An exploratory study of the impact of code smells on software change-proneness. In *2009 16th Working Conference on Reverse Engineering*, pages 75–84. IEEE, 2009.

[17] B Ravi Kiran, Ibrahim Sobh, Victor Talpaert, Patrick Mannion, Ahmad A Al Sallab, Senthil Yogamani, and Patrick Pérez. Deep reinforcement learning for autonomous driving: A survey. *IEEE Transactions on Intelligent Transportation Systems*, 23(6):4909–4926, 2021.

[18] Jens Kober, J Andrew Bagnell, and Jan Peters. Reinforcement learning in robotics: A survey. *The International Journal of Robotics Research*, 32(11):1238–1274, 2013.

[19] Jae Won Lee. Stock price prediction using reinforcement learning. In *ISIE 2001. 2001 IEEE International Symposium on Industrial Electronics Proceedings (Cat. No. 01TH8570)*, volume 1, pages 690–695. IEEE, 2001.

[20] Petro Liashchynskyi and Pavlo Liashchynskyi. Grid search, random search, genetic algorithm: a big comparison for nas. *arXiv preprint arXiv:1912.06059*, 2019.

[21] Hui Liu, Jiahao Jin, Zhifeng Xu, Yanzhen Zou, Yifan Bu, and Lu Zhang. Deep learning based code smell detection. *IEEE transactions on Software Engineering*, 47(9):1811–1837, 2019.

[22] Alberto Maria Metelli. Configurable environments in reinforcement learning: An overview. *Special Topics in Information Technology*, pages 101–113, 2022.

[23] Evangelos Ntentos, Stephen John Warnett, and Uwe Zdun. Supporting architectural decision making on training strategies in reinforcement learning architectures. In *2024 IEEE 21st International Conference on Software Architecture (ICSA)*, pages 90–100. IEEE, 2024.

[24] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Andrea De Lucia. Do they really smell bad? a study on developers' perception of bad code smells. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 101–110. IEEE, 2014.

[25] Python Software Foundation. *ast — Abstract Syntax Trees*, 2026. Accessed: 2026-05-16.

[26] Python Software Foundation. *The Python Language Reference*, 2026. Accessed: 2026-05-16.

[27] Gilberto Recupito, Giammaria Giordano, Filomena Ferrucci, Dario Di Nucci, and Fabio Palomba. When code smells meet ML: on the lifecycle of ML-specific code smells in ML-enabled systems. *Empirical Software Engineering*, 30(5):139, 2025.

[28] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[29] Philip Sedgwick. Pearson's correlation coefficient. *Bmj*, 345, 2012.

[30] SonarSource. Code smells, 2024. Accessed: 2026-05-16.

[31] Ioannis Stamelos, Lefteris Angelis, Apostolos Oikonomou, and Georgios L Bleris. Code quality analysis in open source software development. *Information systems journal*, 12(1):43–60, 2002.

[32] Richard S Sutton, Andrew G Barto, et al. Reinforcement learning. *Journal of Cognitive Neuroscience*, 11(1):126–134, 1999.

[33] Mark Towers, Ariel Kwiatkowski, Jordan Terry, John U Balis, Gianluca De Cola, Tristan Deleu, Manuel Goulao, Andreas Kallinteris, Markus Krimmel, Arjun KG, et al. Gymnasium: A standard interface for reinforcement learning environments. *arXiv preprint arXiv:2407.17032*, 2024.

[34] Eva Van Emden and Leon Moonen. Java quality assurance by detecting code smells. In *Ninth Working Conference on Reverse Engineering, 2002. Proceedings.*, pages 97–106. IEEE, 2002.

[35] Bart Van Oort, Luís Cruz, Maurício Aniche, and Arie Van Deursen. The prevalence of code smells in machine learning projects. In *2021 IEEE/ACM 1st Workshop on AI Engineering-Software Engineering for AI (WAIN)*, pages 1–8. IEEE, 2021.

[36] Ratnadira Widyasari, Zhipeng Yang, Ferdian Thung, Qin Sim, Fiona Wee, Colin Lok, Jonathan Phan, Hoa Qi, Chee Tan, Queenie Tay, and David Lo. NICHE: A curated dataset of engineered machine learning projects in Python. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*, pages 62–66, 2023.

[37] Marco A Wiering and Martijn Van Otterlo. Reinforcement learning. *Adaptation, learning, and optimization*, 12(3):729, 2012.

[38] Aiko Yamashita and Leon Moonen. Do code smells reflect important maintainability aspects? In *2012 28th IEEE international conference on software maintenance (ICSM)*, pages 306–315. IEEE, 2012.

[39] Aiko Yamashita and Leon Moonen. Do developers care about code smells? an exploratory survey. In *2013 20th working conference on reverse engineering (WCRE)*, pages 242–251. IEEE, 2013.

[40] Dapeng Yan, Wenjie Yang, Kui Liu, Zhiming Liu, and Zhikuang Cai. Clean code in practice: Challenges and opportunities. *arXiv preprint arXiv:2507.19721*, 2025.

[41] Haiyin Zhang, Luís Cruz, and Arie Van Deursen. Code smells for machine learning applications. In *Proceedings of the 1st international conference on AI engineering: software engineering for AI*, pages 217–228, 2022.

[42] Ming Zhou, Jun Luo, Julian Villella, Yaodong Yang, David Rusu, Jiayu Miao, Weinan Zhang, Montgomery Alban, Iman Fadakar, Zheng Chen, et al. Smarts: An open-source scalable multi-agent rl training school for autonomous driving. In *Conference on robot learning*, pages 264–285. PMLR, 2021.

Acronyms

AST Abstract Syntax Tree. 10, 28–32, 34, 36, 37, 45, 67

CLI Command-Line Interface. 1, 37, 40

LLM Large Language Model. 33, 67

MDP Markov Decision Process. 4

ML Machine learning. 2, 7–11

RL Reinforcement learning. 1–4, 6–11, 13–19, 21–28, 30–32, 38, 45, 47–56, 58, 61, 63–67

A. Appendix

A.1. Online Sources

Table A.1.: Online sources referenced in this thesis

ID	Title	URL
[S1]	Why Code Quality is Important: The Power of Clean Code	https://www.sonarsource.com/blog/power-of-clean-code/
[S2]	Markov Decision Process (MDP) in Reinforcement Learning	https://www.geeksforgeeks.org/machine-learning/what-is-markov-decision-process-mdp-and-its-relevance-to-reinforcement-learning/
[S3]	DeepMind x UCL: Introduction to Reinforcement Learning (2015) course	https://www.youtube.com/watch?v=2pWv7GOvuf0&list=PLqYmG7hTraZDM-OYHWgPebj2MfCFzFObQ
[S4]	Optuna	https://optuna.org/
[S5]	Streamlit Documentation	https://docs.streamlit.io
[S6]	Pandas: Python Data Analysis Library	https://pandas.pydata.org
[S7]	Plotly Python Graphing Library	https://plotly.com/python/
[S8]	PyMongo Documentation	https://pymongo.readthedocs.io
[S9]	dnspython DNS Toolkit	https://www.dnspython.org
[S10]	OpenAI API Documentation	https://platform.openai.com/docs
[S11]	Importance estimation of hyperparameters in reinforcement learning	https://www.sciencedirect.com/science/article/pii/S0925231225024427

Continued on next page

ID	Title	URL
[S12]	MinimalRL Github	https://github.com/seungeunrho/minimalRL/tree/master
[S13]	Precision and recall	https://developers.google.com/machine-learning/crash-course/classification/accuracy-precision-recall
[S14]	Precision, recall and F1 metrics	http://medium.com/@piyushkashyap045/understanding-precision-recall-and-f1-score-metrics-ea219b908093
[S15]	SumoRL Github	https://github.com/LucasAlegre/sumo-rl
[S16]	PettingZoo Documentation	https://pypi.org/project/pettingzoo/
[S17]	Github API	https://docs.github.com/en/rest
[S18]	AST Python	https://docs.python.org/3/library/ast.html
[S19]	AST Visitor Pattern	https://www.cs.colostate.edu/~cs453/yr2014/Slides/10-AST-visitor.ppt.pdf
[S20]	Source code - rl code smell detector tool	https://github.com/filipgeorgiana/rl-code-smell-detector
[S21]	SumoRL DQN	https://github.com/LucasAlegre/sumo-RL/blob/main/experiments/dqn_big-intersection.py
[S22]	SMARTS	https://github.com/huawei-noah/SMARTS/blob/master/examples/e10_drive/train/run.py
[S23]	PEP8	https://peps.python.org/pep-0008/
[S24]	Intro to Python AST module	https://medium.com/@wshanshan/intro-to-python-ast-module-bbd22cd505f7
[S25]	Introduction to Abstract Syntax Trees in Python	https://earthly.dev/blog/python-ast/
[S26]	PPO-PyTorch Github	https://github.com/nikhilbarhate99/PPO-PyTorch

Continued on next page

ID	Title	URL
[S27]	Papers-in-100-Lines-of-Code	https://github.com/MaximeVandegar/Papers-in-100-Lines-of-Code
[S28]	Reinforcement-learning-an-introduction	https://github.com/ShangtongZhang/reinforcement-learning-an-introduction
[S29]	ICCV2019-LearningToPaint	https://github.com/hzwer/ICCV2019-LearningToPaint
[S30]	Explorations and Exploitation	https://www.geeksforgeeks.org/machine-learning/exploitation-and-exploration-in-machine-learning/
[S31]	Pylint	https://pypi.org/project/pylint/
[S32]	SonarQube	https://www.sonarsource.com/

A.2. Repositories Included in the Empirical Study

Table A.2 lists the 51 reinforcement learning repositories included in the empirical analysis conducted in this thesis.

Table A.2.: Repositories included in the RL code smell analysis

ID	Repository	GitHub URL	Domain
R1	Advanced-Deep-Learning-with-Keras	https://github.com/PacktPublishing/Advanced-Deep-Learning-with-Keras	Deep learning textbook code
R2	DI-engine	https://github.com/opendilab/DI-engine	General RL framework
R3	DRL-Pytorch	https://github.com/XinJingHao/DRL-Pytorch	DRL algorithm implementations
R4	DRL-code-pytorch	https://github.com/Lizhi-sjtu/DRL-code-pytorch	DRL algorithm implementations
R5	DeepRL	https://github.com/ShangtongZhang/DeepRL	Modular DRL library (PyTorch)
R6	DouZero	https://github.com/kwai/DouZero	Card game AI (DouDizhu)

Continued on next page

ID	Repository	GitHub URL	Domain
R7	ElegantRL	https://github.com/AI4Finance-Foundation/ElegantRL	Lightweight DRL framework
R8	HighwayEnv	https://github.com/Farama-Foundation/HighwayEnv	Autonomous driving simulation
R9	ICCV2019-LearningToPaint	https://github.com/hzwer/ICCV2019-LearningToPaint	Generative art / painting agent
R10	JaxMARL	https://github.com/FLAIROx/JaxMARL	Multi-agent RL (JAX)
R11	LightZero	https://github.com/opendilab/LightZero	MCTS + RL benchmark toolkit
R12	MAPDN	https://github.com/Future-Power-Networks/MAPDN	Multi-agent power distribution network
R13	Multi-Agent-Constrained-Policy-Optimisation	https://github.com/chauncygu/Multi-Agent-Constrained-Policy-Optimisation	Safe multi-agent RL
R14	OpenManus-RL	https://github.com/OpenManus/OpenManus-RL	Robotic manipulation
R15	PARL	https://github.com/PaddlePaddle/PARL	High-performance RL framework
R16	PPO-PyTorch	https://github.com/nikhilbarhate99/PPO-PyTorch	PPO algorithm implementation
R17	PantheonRL	https://github.com/Stanford-ILIAD/PantheonRL	Human-agent cooperation
R18	Papers-in-100-Lines-of-Code	https://github.com/Vaibhavs10/Papers-in-100-Lines-of-Code	Minimal RL paper reproductions
R19	RL4LMs	https://github.com/allenai/RL4LMs	RL fine-tuning for language models
R20	SLM-Lab	https://github.com/kengz/SLM-Lab	Modular RL research framework
R21	SMARTS	https://github.com/huawei-noah/SMARTS	Autonomous driving simulation
R22	TensorLayer	https://github.com/tensorlayer/TensorLayer	Deep learning / RL library

Continued on next page

ID	Repository	GitHub URL	Domain
R23	TiKick	`https://github.com/TARTRL/TiKick`	Multi-agent football simulation
R24	acme	`https://github.com/google-deepmind/acme`	General RL framework (DeepMind)
R25	android_env	`https://github.com/google-deepmind/android_env`	Android device RL environment
R26	bindsnet	`https://github.com/BindsNET/bindsnet`	Spiking neural network simulation
R27	chatarena	`https://github.com/Farama-Foundation/chatarena`	Multi-agent language game env
R28	deep-neuroevolution	`https://github.com/uber-research/deep-neuroevolution`	Neuroevolution algorithms
R29	dm_control	`https://github.com/google-deepmind/dm_control`	Physics-based locomotion control
R30	evotorch	`https://github.com/nnaisense/evotorch`	Evolutionary computation library
R31	football	`https://github.com/google-research/football`	Football / soccer simulation env
R32	godot_rl_agents	`https://github.com/edbeeching/godot_rl_agents`	RL in Godot game engine
R33	gym-pybullet-drones	`https://github.com/utiasDSL/gym-pybullet-drones`	Multi-drone flight simulation
R34	macad-gym	`https://github.com/praveen-palanisamy/macad-gym`	Multi-agent CARLA driving env
R35	maro	`https://github.com/microsoft/maro`	RL for resource optimisation
R36	meltingpot	`https://github.com/google-deepmind/meltingpot`	Multi-agent social interaction env
R37	minimalRL	`https://github.com/seungeunrho/minimalRL`	Minimal RL algorithm demos

Continued on next page

ID	Repository	GitHub URL	Domain
R38	og-marl	`https://github.com/instadeepai/og-marl`	Offline multi-agent RL datasets
R39	pymarl3	`https://github.com/tjuHaoXiaotian/pymarl3`	Cooperative MARL (StarCraft)
R40	pysc2	`https://github.com/google-deepmind/pysc2`	StarCraft II RL environment
R41	pytorch-a2c-ppo-acktr-gail	`https://github.com/ikostrikov/pytorch-a2c-ppo-acktr-gail`	Policy gradient implementations
R42	reinforcement-learning-an-introduction	`https://github.com/ShangtongZhang/reinforcement-learning-an-introduction`	Sutton & Barto textbook code
R43	rl-baselines3-zoo	`https://github.com/DLR-RM/rl-baselines3-zoo`	Trained baselines & benchmarks
R44	rlcard	`https://github.com/datamllab/rlcard`	Card game RL environment
R45	rlkit	`https://github.com/rail-berkeley/rlkit`	Off-policy RL algorithms
R46	smac	`https://github.com/oxwhirl/smac`	StarCraft multi-agent challenge
R47	sumo-rl	`https://github.com/LucasAlegre/sumo-rl`	Traffic signal control (SUMO)
R48	tensorpack	`https://github.com/tensorpack/tensorpack`	TensorFlow training framework
R49	tf2multiagentrl	`https://github.com/JohannesAck/tf2multiagentrl`	Multi-agent RL (TensorFlow 2)
R50	tnt	`https://github.com/meta-pytorch/tnt`	PyTorch training utilities
R51	xuance	`https://github.com/agi-brain/xuance`	Comprehensive DRL library