INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA INFORMÁTICA




Systematic Review of Exploration Strategies and Evaluation Metrics in Reinforcement Learning using an Automated Software Testing Solution

AFONSO MEIRELES BERTÃO
Junho de 2025

POLITÉCNICO
DO PORTO

ISEP INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

Systematic Review of Exploration Strategies and Evaluation Metrics in Reinforcement Learning using an Automated Software Testing Solution

Afonso Meireles Bertão

Dissertation

Master's Degree in

Informatics Engineering - Software Engineering

Supervisor: Dr. Isabel Cecília Correia da Silva Praça Gomes Pereira

Co-supervisor: Eng. Tiago Fontes Dias

Porto, June 2025

Statement of Integrity

I hereby declare that I have conducted this academic work with integrity.

I have not plagiarized or applied any form of undue use of information or falsification of results along the process leading to its elaboration.

Therefore, the work presented in this document is original and authored by me, having not previously been used for any other end.

I further declare that I have fully acknowledged the Code of Ethical Conduct of P. PORTO.

ISEP, Porto, 25 de June de 2025

Dedicatory

I want to dedicate this thesis and all my academic work to my family, girlfriend and friends that always supports me through all the steps I take in my life. For them all the hard work is worth it.

Abstract

In an era defined by rapid technological evolution, software systems underpin essential domains such as healthcare, finance and digital communication. Central to these systems are REST APIs, which enable seamless interaction across distributed services. However, the increasing complexity and ubiquity of these APIs have elevated concerns surrounding their security and robustness. Vulnerabilities in REST APIs can result in severe consequences, including data breaches, service outages and compromised user trust highlighting the need for intelligent and adaptive testing solutions.

This thesis explores the integration of reinforcement learning (RL) into automated software testing, with a focus on algorithmic exploration strategies and evaluation metrics applied within FuzzTheRest, a state-of-the-art REST API fuzzing tool. While FuzzTheRest leverages the Epsilon-Greedy exploration strategy for action selection, its simplistic balance between exploration and exploitation limits its effectiveness in navigating complex API environments and uncovering subtle security flaws. To address these limitations, this review investigates the potential of Boltzmann Exploration, a probabilistic approach that adjusts action selection based on a softmax distribution over Q-values, enabling more nuanced and informed exploration of the test space.

The study conducts a comparative analysis of multiple input generation strategies including random, evolutionary and RL-driven techniques evaluating their performance in adaptation to APIs and endpoints with complex parameterization terms, successful total number of HTTP responses, exploration fuzzing metrics and computational efficiency. In particular, the review highlights the impact of exploration strategies like Epsilon-Greedy and Boltzmann on the quality and depth of fuzzing outcomes.

By synthesizing current literature and experimental insights, this review lays the groundwork for advancing RL-guided fuzzing methodologies. The findings aim to empower developers, testers and security practitioners with a deeper understanding of how exploration strategies choices influence the effectiveness of automated testing tools, contributing to more secure and resilient software systems.

Keywords: Reinforcement Learning, Fuzzing, REST APIs, Epsilon-Greedy, Boltzmann Exploration, Exploration Strategies, Automated Software Testing, Fuzzing Metrics, FuzzTheRest

Resumo

Num contexto de rápida evolução tecnológica, os sistemas de software sustentam domínios essenciais como a saúde, as finanças e a comunicação digital. No centro destes sistemas encontram-se as APIs REST, que permitem a interação fluida entre serviços distribuídos. Contudo, a crescente complexidade e generalização destas APIs têm levantado sérias preocupações relativamente à sua segurança e robustez das mesmas. Vulnerabilidades em APIs REST podem resultar em consequências graves, como violações de dados, interrupções de serviço e perda de confiança por parte dos utilizadores o que sublinha a necessidade de soluções de teste inteligentes e adaptativas.

Esta revisão sistemática explora a integração de técnicas de aprendizagem por reforço (Reinforcement Learning) no domínio dos testes automatizados de software, com foco nas estratégias de exploração de seleção de ações e métricas de avaliação aplicadas na ferramenta FuzzTheRest, uma solução avançada de fuzzing de APIs REST. Embora o FuzzTheRest utilize a estratégia de exploração Epsilon-Greedy na seleção de ações, a sua abordagem simplista na gestão do equilíbrio entre exploração e exploração limita a eficácia na navegação por ambientes API complexos e na deteção de vulnerabilidades subtis. Para resolver estas limitações, esta tese investiga o potencial da Exploração de Boltzmann, uma abordagem probabilística que ajusta a seleção de ações com base numa distribuição softmax dos valores Q, que permite uma exploração mais informada e precisa do espaço de testes.

É realizada uma análise comparativa de diversas estratégias de geração de entradas que inclui abordagens aleatórias, evolutivas e baseadas em aprendizagem por reforço ao avaliar o seu desempenho na adaptação a APIs e endpoints com parametrização complexa, eficácia nas respostas HTTP em diversos cenários, métricas de exploração relacionadas a fuzzing e eficiência computacional. Em particular, destaca-se o impacto de estratégias de exploração como o Epsilon-Greedy e a Exploração de Boltzmann na qualidade e profundidade dos resultados de fuzzing.

Ao sintetizar literatura atual e perceções experimentais, esta revisão estabelece as bases para o avanço de metodologias de fuzzing orientadas por aprendizagem por reforço. Os resultados visam fornecer aos programadores, testadores e profissionais de segurança uma compreensão mais aprofundada sobre como a escolha de estratégias de exploração influencia a eficácia das ferramentas de teste automatizado, contribuindo para sistemas de software mais seguros e resilientes.

Palavras-chave: Aprendizagem por Reforço, Fuzzing, APIs REST, Epsilon-Greedy, Exploração de Boltzmann, Estratégias de Exploração, Testes Automatizados de Software, Métricas de Fuzzing, FuzzTheRest

Acknowledgments

I would like to thank my family for all the support given to me through all my life and my academic work, they are the basis of everything I do. My Dad, Mother, Sister and Grandmother who are with me every day and keep the balance and stability in my life, I work hard every day for them to give them everything they need me to be. They are truly the best I could ever ask for.

Thanks to my girlfriend Bia, who has been the greatest supporter and companion I could ever dream of having. Gives me the motivation and the discipline to work hard every day to reach my objectives and be accountable for my actions always. I am a better person and a better man by her side.

Also want to thank Supervisors Dr. Isabel Praça and Eng. Tiago Dias for their support during the making of this thesis, for their detailed revision of this document and helping it be the best it can be.

Index

List of Figures

List of Tables

Abbreviations

ACM	Association for Computing Machinery
AFL	American Fuzzy Lop
AI	Artificial Intelligence
API	Application Programming Interface
API SUT	Application Programming Interface System Under Test
DL	Deep Learning
EC	Exclusion Criteria
GECAD	Grupo de Investigação em Engenharia e Computação Inteligente para a Inovação e o Desenvolvimento
HTTP	Hypertext Transfer Protocol
IC	Inclusion Criteria
IEEE	Institute of Electrical and Electronics Engineers
IPP	Instituto Politécnico do Porto
ISEP	Instituto Superior de Engenharia do Porto
ML	Machine Learning
OWASP	Open Web Application Security Project
PRIMA	Preferred Reporting Items for Systematic Reviews and Meta-Analyses
REST	Representational State Transfer
RL	Reinforcement Learning
RQ	Research Question
UCB	Upper Confidence Bound (Exploration Strategy)

1 Introduction

This chapter introduces the problem addressed in this thesis, outlining its motivation, context and relevance in the current technological landscape. It also presents the research questions, objectives and contributions of the study, as well as the structure of the document

1.1 Contextualization

In today's highly interconnected digital world, software systems underpin a wide range of sectors, including healthcare, finance, education and entertainment. As these systems grow in complexity and scale, their security and robustness become increasingly critical [1]. One of the foundational elements of modern web-based software architecture is the Application Programming Interface (API), which facilitates communication between different components and services. RESTful APIs have become the de facto standard for enabling scalable and interoperable web services [2].

However, the widespread exposure of APIs, especially those accessible via the public internet, introduces a significant attack surface. Poorly designed or inadequately tested APIs are frequently exploited by malicious actors, resulting in data breaches, service interruptions and systemic vulnerabilities [3]. According to recent security reports, API-related attacks are on the rise, primarily due to weak authentication, improper input validation and misconfigurations [4].

Fuzz testing (or fuzzing) has emerged as an effective technique for uncovering security flaws by injecting large volumes of unexpected or malformed inputs into a system to observe anomalous behaviors [5]. Specifically, fuzzing for RESTful APIs has gained attention as a means of validating input handling, authentication flows and endpoint behavior [6]. Tools such as RESTler [7] and FuzzTheRest [8] have been developed to automate this process. However, traditional fuzzers often rely on static or random input generation strategies, which may fail to comprehensively explore complex API paths or trigger deep-seated vulnerabilities [9].

To overcome these limitations, researchers have begun integrating Machine Learning (ML) and Reinforcement Learning (RL) techniques into fuzzing workflows. RL allows a fuzzer to learn from feedback obtained during the fuzzing process guiding input generation decisions to optimize vulnerability discovery [10]. This dynamic, feedback-driven approach can significantly increase test efficiency, especially when applied to high-dimensional and stateful API environments [11].

This thesis is built upon an existing RESTful API fuzzer, FuzzTheRest, which incorporates basic RL techniques to guide test case generation. The goal of this research is to enhance the fuzzers capability by evaluating and integrating advanced RL exploration strategies, specifically comparing Epsilon-Greedy and Boltzmann Exploration strategies. Furthermore, this work explores the introduction of novel evaluation metrics, including those based on HTTP status codes and endpoint coverage, to improve the classification and analysis of discovered vulnerability.

Through this enhancement, the thesis aims to contribute to the field of automated software testing by advancing intelligent fuzzing methodologies that are both adaptive and efficient, ultimately aiding developers and security professionals in identifying and mitigating threats more effectively.

1.2 Problem Description

With the proliferation of distributed systems and service-oriented architectures, RESTful APIs have become fundamental components of modern software applications [12]. Their role as communication channels between microservices, mobile applications and client interfaces places them in a critical position within system architecture. As a result, they are often targeted by attackers aiming to exploit poorly validated inputs, flawed authentication mechanisms or insecure configurations [13].

Although traditional testing techniques such as fuzzing have proven valuable in exposing certain types of flaws, the methods used in most existing API fuzzers are limited in their adaptability and depth. These tools commonly employ static, rule-based or purely random input generation methods, which lack the sophistication to efficiently explore the diverse and complex state space of modern APIs [14]. Consequently, vulnerabilities that depend on specific input sequences or nuanced system states may go undetected.

Reinforcement Learning provides a promising avenue for addressing these challenges. By enabling the fuzzer to learn from the outcomes of previous actions such as which inputs triggered unexpected behavior RL allows for adaptive input generation, improving the likelihood of discovering critical flaws over time [15]. Despite this potential, there remains a notable research gap concerning the best practices for incorporating RL into API fuzzing. Specifically, the choice of exploration strategies (such as Epsilon-Greedy vs. Boltzmann) and the design of reward mechanisms have a profound impact on the efficiency and accuracy of the fuzzing process, yet they remain underexplored [16].

This thesis addresses these gaps by systematically studying and implementing RL-based action selection strategies within FuzzTheRest, focusing on improving its ability to uncover vulnerabilities in RESTful APIs. In addition, the research investigates novel classification and performance metrics, including dynamic feedback from HTTP responses (e.g., status codes, error messages) and endpoint traversal paths, to better assess the quality of the fuzzing process.

The proposed enhancements aim to not only improve fuzzing effectiveness but also contribute to the broader understanding of how intelligent exploration strategies and data-driven strategies can transform API security testing. The work aspires to deliver practical insights and tools that strengthen the security posture of modern web systems.

1.3 Collaboration with GECAD Research Unit

The development of this project within an academic setting specifically in collaboration with the GECAD Research Unit [17] at the Instituto Superior de Engenharia do Porto (ISEP) [18] was strategically motivated by the opportunity to engage with a forward-thinking research environment focused on innovation, scientific rigor and technological advancement.

GECAD is internationally recognized for its contributions to intelligent systems, decision support and applied artificial intelligence, particularly in energy, cybersecurity and intelligent software systems. Its multidisciplinary team provides access to advanced scientific methodologies, cutting-edge research tools and a strong emphasis on Reinforcement Learning (RL) and intelligent testing methods are key to addressing the technical challenges of this thesis.

This partnership fostered a collaborative ecosystem where applied research could be enhanced by theoretical foundations and academic supervision, while also benefiting from access to experimental infrastructures, peer feedback and a dynamic knowledge-sharing environment. In turn, GECAD researchers and students gained practical experience in the field of automated software testing and API security, by contributing to the development, experimentation and validation of the proposed improvements to FuzzTheRest.

Moreover, the partnership exemplifies a mutually beneficial university-industry collaboration model, where academia supports technological innovation and industry provides real-world challenges that guide relevant research. The integration of student-led research into GECAD's broader scientific agenda allowed for the continuous refinement of ideas through critical analysis and experimentation, ultimately enhancing both academic learning outcomes and the project's practical impact.

This synergy between the academic and professional spheres contributes to the development of high-impact, research-driven solutions that address concrete problems in API security and software robustness. It strengthens the commitment of both parties to advancing the state of the art in software engineering and cybersecurity, while also cultivating essential skills in innovation, experimentation and applied AI.

1.4 Research Questions and Objectives

This thesis aims to improve and extend an existing RESTful API fuzzer, which currently employs a Reinforcement Learning (RL) algorithm to guide the generation of inputs. To address the challenges associated with the automation of software testing and vulnerability detection for RESTful APIs, four Research Questions (RQ) have been formulated to guide the investigation:

Research Questions (RQ):

- RQ1: What are the limitations and flaws in the current generation of RESTful API fuzzers that use Reinforcement Learning for input generation?
- RQ2: What additional metrics and techniques can be explored to improve the analysis and testing capabilities of RESTful API fuzzers?
- RQ3: How do different exploration strategies, including Boltzmann Exploration, compare to traditional algorithms (such as Epsilon-Greedy) in terms of their effectiveness in generating diverse and targeted inputs for discovering vulnerabilities in RESTful APIs?
- RQ4: How can the exploitation-exploration balance and reward-based learning metrics, such as the exploitation vs. exploration ratio and cumulative reward per episode, be used to enhance the exploration of API structures and improve vulnerability detection during API testing?

The primary objective of this thesis is to enhance and expand the existing RESTful API fuzzer by integrating innovative techniques and advanced algorithms for test case generation, with a primary focus on Reinforcement Learning. The specific objectives (O) are outlined as follows:

Objectives (O):

- O1: Investigate the current state of the art in RESTful API fuzzing, with a focus on the limitations of traditional exploration strategies (such as Epsilon-Greedy) and the potential advantages of Boltzmann Exploration in improving test case generation for vulnerability detection.
- O2: Identify and address the inefficiencies in current fuzzing tools by evaluating how Boltzmann Exploration and new RL-based metrics (such as the exploitation vs. exploration ratio and cumulative reward per episode) can enhance the accuracy and efficiency of RESTful API testing.
- O3: Develop enhancements to the existing RL-based fuzzer by incorporating Boltzmann Exploration to better balance exploration and exploitation and integrate new metrics to assess the effectiveness of test case generation, exploration behavior and vulnerability detection.

- O4: Extend the functionality of the fuzzer by utilizing Boltzmann Exploration to guide more adaptive and intelligent exploration of API state spaces and employ new metrics to refine vulnerability detection strategies and provide deeper insights into the fuzzers exploration and learning processes.

The goal of this project is to develop a new version of the existing fuzzer, expanding its capabilities both technically and in terms of software engineering, with a focus on improving the discovery and detection of vulnerabilities in RESTful APIs.

1.5 Project Planning

The thesis follows an Agile-based approach [19], allowing continuous updates and flexibility. Each phase builds on the previous one and helps shape the final solution with regular feedback and refinement.

1. **Problem Analysis**: This phase identifies the weaknesses in current RESTful API fuzzers, especially those using basic methods like Epsilon-Greedy. Real-world cases and reports from sources like OWASP show the need for smarter fuzzing that can handle complex API structures.
2. **Goal Definition**: The main goal is to improve an existing API fuzzer by adding Reinforcement Learning with Boltzmann Exploration and new metrics like cumulative reward per episode and exploration/exploitation ratios. These changes aim to increase input diversity and make vulnerability detection more efficient.
3. **State of the Art and Related Work**: This step reviews current research and tools like RESTler, ML-based fuzzers and RL-guided approaches. Few explore Boltzmann strategies or use dynamic rewards, which is the gap this project addresses.
4. **Solution Design**: The new fuzzer is planned with a modular structure. It includes a learning agent, custom rewards and a smarter input generator using Boltzmann Exploration.
5. **Solution Implementation and Evaluation**: The system is built in steps. Boltzmann replaces Epsilon-Greedy with better control over input selection. The evaluation tests it on public APIs, checking how well it detects issues and explores API paths.
6. **Documentation**: All phases are documented continuously. This includes design choices, test setups, code changes and evaluation results, ensuring transparency and easier future development.

1.6 Ethical Considerations

This thesis follows to a set of ethical guidelines based on the IPP Code of Conduct for students [20]. The principles outlined were carefully observed throughout the project. Article 6 was closely followed, especially statement 2.8, which prohibits the use of ideas, sentences, paragraphs or texts without proper citation or referencing. This rule was strictly respected. Statement 2.9, which emphasizes the importance of not presenting previously published or displayed work as original without appropriate acknowledgment, was also fully observed. Furthermore, statement 2.10, which warns against presenting collaborative work without the consent of all contributors involved, was respected and properly acknowledged the contributions made. In addition, the guidance of statement 2.11, which advises against the use of falsified or misleading results, was diligently followed.

The commitment defined in Article 8 of the IPP Code was upheld throughout the project. This includes maintaining academic integrity, ensuring originality and adhering to the ethical standards. The guidelines stated in Article 10 were also respected. These include conducting research with care and thorough analysis, properly citing and referencing relevant works and presenting results in a clear and reproducible manner.

As a student enrolled in the master's program in Informatics Engineering with a specialization in Software Engineering, I also followed the IEEE Code of Ethics [21]. Core principles such as honesty, integrity and a dedication to truthfulness were incorporated throughout the development of this work. Efforts were made to avoid conflicts of interest, whether due to personal bias or external influences, to always maintain objectivity and fairness.

1.7 Document Structure

This document is structured into seven chapters, each designed to guide the reader through the progression of the research project from its motivation and planning to the design, implementation and evaluation of the proposed solution.

Chapter 1 - Introduction:

This chapter provides the foundational context for the thesis, outlining the motivation, research questions and objectives that guided the work. It introduces the challenges of securing RESTful APIs and highlights the significance of fuzz testing and Reinforcement Learning (RL) in identifying vulnerabilities. Furthermore, it explains the impact of optimizing fuzzing techniques and how the proposed enhancements can contribute to stronger software security systems.

Chapter 2 - State of the Art:

This chapter reviews existing literature and tools in the domain of automated software testing for RESTful APIs, with particular emphasis on fuzzing techniques and RL-based input generation. It explores how fuzzers have evolved, from randomized input generators to intelligent systems guided by machine learning. Key limitations of current technologies are identified and Boltzmann Exploration is introduced as a promising alternative to traditional strategies such as Epsilon-Greedy.

Chapter 3 - Existing System Analysis:

This chapter analyzes current fuzzing systems used in the automated testing of RESTful APIs. It investigates existing methodologies that utilize Reinforcement Learning, discussing their capacity to detect edge-case vulnerabilities. The performance, scalability and generalization capabilities of tools are evaluated to understand the gaps that the proposed solution aims to bridge. These insights form the baseline for the improvements introduced in subsequent chapters.

Chapter 4 - Design of the Solution:

This chapter describes the architecture and design rationale of the enhanced RESTful API fuzzer. It elaborates on the selected RL algorithms, particularly focusing on the integration of Boltzmann Exploration and introduces custom reward-based metrics such as exploitation/exploration ratio and cumulative reward tracking. These techniques are used to guide the fuzzer through complex input spaces more effectively. Design decisions, modular structure and technical constraints are discussed in depth.

Chapter 5 - Implementation of the solution:

This chapter details the technical implementation of the enhanced reinforcement learning-based RESTful API fuzzer. It focuses on how the core components particularly the Q-Learning agent, Boltzmann exploration strategy, action selection mechanisms and reward-based feedback loops were developed and integrated into the testing framework. In addition, it outlines how APIs were selected for testing, how mutation strategies were applied, and how learning metrics were captured, logged, and visualized.

Chapter 6 - Tests and Solution Evaluation:

This chapter presents the experimental framework and the evaluation methodology. The enhanced fuzzer, now integrating Boltzmann Exploration and additional RL-based metrics, is tested on a public APIs with real word data. Special attention is given to how the exploitation vs. exploration ratio and cumulative reward per episode influence input generation and vulnerability exposure. Comparative testing with traditional fuzzers highlights the improvements achieved through adaptive learning strategies. The chapter also explores the

fuzzers' ability to navigate complex API state spaces and interprets the results to identify opportunities for further optimization and generalization across diverse API architectures.

Chapter 7 - Conclusion:

The final chapter provides a summary of the thesis contributions, emphasizing the advancements achieved through the integration of Boltzmann-based exploration and the development of novel RL-driven metrics into the RESTful API fuzzing process. The experimental results demonstrate notable improvements in vulnerability detection, learning efficiency and input generation strategy compared to conventional methods. The chapter reflects on how these enhancements enable more intelligent and context-aware fuzzing for modern APIs. The chapter concludes by highlighting the long-term potential of Reinforcement Learning to redefine how automated security testing tools evolve in response to increasingly complex and dynamic software ecosystems.

2 State of the Art

This chapter conducts a comprehensive review of the current state of automatic software testing tools, particularly focusing on fuzzer techniques for identifying vulnerabilities in APIs. The review is structured around three main objectives: defining the flexibility of API fuzzing tools, identifying patterns and methodologies that enhance the effectiveness of fuzz testing and evaluating the performance of these tools in detecting security flaws and vulnerabilities in RESTful APIs. Special attention is given to advanced fuzzing strategies that incorporate reinforcement learning (RL), particularly the use of Boltzmann Exploration as an alternative to traditional methods like Epsilon-Greedy. Additionally, the integration of new RL-based performance metrics such as the cumulative reward per episode and the exploration vs. exploitation ratios explored to assess and refine the effectiveness of fuzzers in discovering complex vulnerabilities and improving their adaptability to different API structures.

2.1 Automated Software Testing

Automated software testing [22] is a cornerstone of modern software development, providing essential mechanisms to ensure the quality, reliability and security of increasingly complex and interconnected systems. As software systems scale in size and sophistication, traditional manual testing approaches are no longer viable due to their time-consuming nature and limited scope. Automated testing leverages advanced tools and methodologies to systematically identify defects, vulnerabilities and performance bottlenecks, enabling faster development cycles and higher software reliability. This chapter discusses the principles and advancements of automated software testing, emphasizing its role in stress-testing and vulnerability detection, with a focus on fuzz testing (fuzzing) and its evolution through modern approaches such as machine learning and artificial intelligence.

2.1.1 Overview of Automated Software Testing

Automated software testing encompasses techniques that execute tests, validate results and detect faults without human intervention. These approaches aim to optimize test efficiency, coverage and accuracy while reducing time and resource costs. Among various methodologies, fuzz testing (fuzzing) has emerged as a particularly effective strategy for stress-testing software and discovering security vulnerabilities.

Fuzzing is a widely used automated testing technique that involves generating a large volume of inputs typically random or semi-random to uncover bugs [23], vulnerabilities or unexpected behaviors. It is particularly valuable in identifying edge cases that are difficult to detect through conventional testing methods.

Modern fuzzing techniques have evolved to address its traditional limitations, incorporating strategies such as mutation-based fuzzing [24], which modifies existing inputs to create variations that can explore new code paths or trigger unexpected behaviors; generation-based fuzzing [25], which generates inputs from scratch using predefined rules or specifications, making it suitable for structured data formats and protocols; and coverage-guided fuzzing [26], which uses feedback from code execution (e.g., code coverage metrics) to guide input generation and optimize exploration of the software's execution space. Tools like AFL (American Fuzzy Lop) [27] and libFuzzer [28] exemplify this approach. While fuzzing has proven highly effective, traditional methods often struggle with complex systems, such as RESTful APIs, where input interactions and system states are interdependent. Addressing these challenges requires leveraging cutting-edge techniques, including machine learning and reinforcement learning.

2.1.2 Role of Machine Learning and AI in Software Testing

The integration of machine learning (ML) [29] and artificial intelligence (AI) [30] into software testing has marked a significant advancement in the field. These technologies enable automated systems to intelligently generate and prioritize test cases, adapt to dynamic software environments and identify subtle vulnerabilities. Reinforcement learning is particularly promising in advancing fuzzing techniques. RL models are trained to optimize a reward signal, such as discovering new code paths, exposing vulnerabilities or maximizing system interaction coverage. Key features of RL-based fuzzers include adaptive input generation [31], where inputs are dynamically adjusted based on feedback from the system under test, exploration vs. exploitation trade-off [32], which balances discovering new areas of the system (exploration) with focusing on known areas of interest (exploitation) and complex state exploration, where RL models manage stateful systems like RESTful APIs, understanding sequences of calls and dependencies. Real-world examples such as tools like AFL++ demonstrate how RL improves fuzzing efficiency and coverage.

2.1.3 Principles of Fuzzing in Automated Software Testing

Fuzzing or fuzz testing is an automated technique that involves feeding random or semi-random data into a program to identify bugs and vulnerabilities that could be exploited. The core principle of fuzzing is that by stressing the system with unexpected or malformed inputs, testers can discover edge cases and failures that are typically hard to find using traditional testing methods. The fuzzing process is characterized by input generation [33], where the fuzzer generates a large volume of inputs, often random or semi-random, to explore different execution paths of the system. The inputs can range from simple strings to complex data structures or protocol messages. Fault detection [34] follows, as the fuzzer monitors the system's response to inputs, looking for crashes, memory leaks or unexpected behaviors such as assertion failures or incorrect output. An advanced feedback mechanism, such as coverage-guided fuzzing, uses feedback from the execution environment (like code coverage metrics) to guide the generation of new inputs, enhancing the efficiency of the fuzzing process.

2.2 Artificial Intelligence

Artificial Intelligence (AI) [35] has become a transformative force in many domains, including software testing. In the context of automated software testing, AI plays a pivotal role in enhancing the accuracy, efficiency and scope of testing processes. By enabling systems to learn from data, adapt to dynamic environments and make decisions based on complex patterns, AI methods are revolutionizing how software is tested, especially in areas such as vulnerability detection, test case generation and system behavior prediction.

This chapter explores the principles and applications of AI in automated software testing, with a particular focus on machine learning (ML), deep learning (DL) [36] and reinforcement learning (RL) examining how these AI techniques are integrated into fuzz testing and other testing methodologies, their impact on software quality assurance (QA) [37] and the prospects of AI-driven testing solutions.

2.2.1 Overview of Artificial Intelligence in Software Testing

Artificial Intelligence in software testing refers to the use of AI techniques such as machine learning, deep learning and reinforcement learning to automate various aspects of the testing process. AI can assist in test planning, test case generation, defect prediction and anomaly detection, among other tasks. One of the most significant contributions of AI to software testing is its ability to handle vast amounts of data and learn from past testing experiences to improve future test cycles.

In traditional automated testing, predefined scripts and patterns are used to verify software behavior. However, AI introduces the ability for systems to autonomously adapt to changing environments, learn from historical test results and dynamically generate new test scenarios.

This leads to smarter testing tools that require less manual intervention, provide better coverage and can uncover previously undetectable issues. Several AI techniques are transforming software testing. Below are some of the most used AI methods and their applications in automated testing.

2.2.2 Machine learning (ML)

Machine learning [38] is a subset of AI that focuses on algorithms that allow systems to learn from data and make decisions without explicit programming. In the context of software testing, ML can be used to predict which areas of the software are most likely to contain defects, optimize test case selection and adapt testing strategies over time. ML models can analyze historical test data and automatically generate new test cases that have a higher likelihood of exposing defects. These models can identify patterns in previous test failures and prioritize areas of the application that need testing. ML algorithms can also predict the likelihood of defects in a piece of code based on historical data. By analyzing factors such as code complexity, recent changes and past defects, ML models can focus testing efforts on the area's most prone to errors, improving testing efficiency.

2.2.3 Deep learning (DL)

Deep learning [39], a more advanced subset of ML, uses artificial neural networks with many layers to process vast amounts of data and learn from complex patterns. In software testing, deep learning techniques can enhance automated test generation, anomaly detection and natural language processing for test specification. Deep learning can be used to generate complex test cases by learning patterns from the codebase. This is especially useful in situations where traditional rule-based systems struggle to generate valid test inputs. Deep learning models can also detect unusual behavior in the system, such as unexpected system responses or performance bottlenecks, by learning the normal behavior over time and flagging deviations from it.

2.2.4 Reinforcement learning (RL)

Reinforcement learning [40] is an AI technique where an agent learns to make decisions by interacting with the environment and receiving feedback in the form of rewards or penalties. In the context of software testing, RL can optimize the testing process by dynamically adjusting inputs based on feedback from the system under test (SUT). RL can adapt testing strategies by learning from previous testing cycles. It can determine which test paths should be explored further and which can be deprioritized, allowing for more efficient testing and better test coverage. In fuzz testing, RL techniques can guide the fuzzing process by dynamically adjusting the input generation based on feedback. This makes fuzz testing more efficient by focusing on inputs that are more likely to uncover vulnerabilities.

2.2.5 AI in Fuzz Testing

Fuzz testing (fuzzing) is a popular method for stress-testing software and discovering vulnerabilities. AI has begun to enhance fuzzing techniques, making them more adaptive and efficient. By incorporating AI into fuzz testing, the process becomes more targeted, potentially uncovering hidden vulnerabilities that traditional fuzzing methods might miss. Traditional fuzzers generate random or semi-random inputs and monitor system responses. However, AI-enhanced fuzzers use machine learning models to adapt the input generation process dynamically. These models can learn which inputs are more likely to uncover vulnerabilities based on feedback from previous test runs. AI techniques, particularly reinforcement learning, enable fuzzers to adapt inputs in real-time based on the results of previous tests. For instance, if a certain input uncovers a bug, the fuzzer can generate variations of that input to explore deeper code paths, increasing the likelihood of discovering additional vulnerabilities. Stateful systems, such as RESTful APIs, require maintaining a sequence of interactions and the order of inputs can affect the system's state. AI techniques can be used to model the stateful interactions and generate more realistic test cases that exercise the system's state transitions, leading to more effective testing of complex systems. Coverage-guided fuzzers use feedback from the code execution, such as code coverage metrics, to guide input generation. By integrating AI models, fuzzers can optimize this process further by intelligently prioritizing areas of the code that are underexplored, improving both the coverage and the chances of discovering critical vulnerabilities.

2.2.6 AI for Test Optimization

In addition to enhancing specific testing techniques like fuzzing, AI can optimize the overall testing process by making smarter decisions regarding test case selection, prioritization and execution. Traditional test suites can become overly large, making it difficult to execute every test case in a reasonable amount of time. AI can be used to prioritize test cases that are more likely to uncover defects or provide valuable insights, improving the efficiency of the testing process. AI can also help minimize the size of the test suite by eliminating redundant test cases. By learning which tests are likely to cover similar functionality, AI can reduce the number of tests required without compromising the quality of the testing process. Over time, as software evolves, test suites can become outdated or ineffective. AI can predict when test suites need to be updated based on changes in the codebase, ensuring that the tests remain relevant and useful. In conclusion, AI is set to revolutionize automated software testing by making it more efficient and more resilient in the face of modern development challenges. As these technologies mature, they will play an indispensable role in ensuring that software remains secure, reliable and performant, ultimately supporting the successful deployment of high-quality software systems in a rapidly changing technological landscape.

2.3 Reinforcement Learning (RL)

Reinforcement Learning (RL) represents an AI paradigm in which an agent learns how to behave in an environment by performing actions and receiving feedback in the form of rewards or penalties. Unlike supervised learning, where the correct answers are provided, RL focuses on exploration and exploitation to discover optimal strategies. In the context of automated software testing, RL is used to optimize testing strategies dynamically and to explore untested code paths, leading to more comprehensive and adaptive testing processes.

Reinforcement Learning is unique in its approach to learning, relying on trial-and-error methods where an agent interacts with its environment and adjusts its behavior based on the outcomes of those interactions. It is particularly valuable in testing scenarios that involve dynamic, changing environments or systems where traditional testing methods might not be sufficient. RL agents learn optimal behaviors through exploration and experimentation, which is beneficial when traditional methods cannot easily identify the best course of action. Additionally, RL agents adapt strategies based on real-time feedback from the environment, making them flexible in complex, changing situations. The agent's primary objective is to maximize cumulative rewards over time, aligning with the goal of discovering defects efficiently in software testing. Moreover, RL agents can dynamically adjust to new situations, ensuring their decisions remain relevant even as the software being tested evolves.

Reinforcement Learning is a powerful tool in automated software testing because of its ability to adapt and optimize test strategies based on feedback from the system. It can significantly improve test coverage, efficiency and vulnerability detection by dynamically adjusting testing strategies. RL can adjust testing strategies in real-time based on feedback from the software being tested, ensuring optimal test coverage. It is particularly useful in fuzz testing, where RL generates inputs that are more likely to uncover vulnerabilities, thereby enhancing the testing process. RL agents can also explore untested code paths, increasing the likelihood of uncovering defects in less-explored areas of the system. Additionally, RL is adept at modeling stateful interactions, such as API sequences, ensuring that tests reflect real-world usage patterns and state transitions.

While RL offers many advantages, it also presents several challenges. These include the complexity of modeling environments, balancing exploration and exploitation, high computational demands and issues with sparse rewards in some testing scenarios. Accurately modeling the software environment for RL training can be difficult, especially for complex systems. RL must also strike a balance between exploring new, untested paths and exploiting known paths to maximize the discovery of defects. Training RL agents requires significant computational resources, particularly when testing complex software environments. Furthermore, in certain testing scenarios, rewards for uncovering defects may be infrequent, which can slow down the learning process.

The use of Reinforcement Learning in automated software testing brings numerous benefits, particularly in optimizing testing strategies, improving efficiency and increasing coverage. RL adapts testing strategies to thoroughly explore software, increasing the likelihood of

discovering defects that may have been missed by traditional methods. By focusing resources on high-reward areas, RL improves testing efficiency and minimizes wasted effort. RL also adapts continuously to changes in the software, ensuring that testing strategies remain relevant as the software evolves. Finally, RL's focus on high-risk areas increases the likelihood of uncovering vulnerabilities, improving software security and reliability.

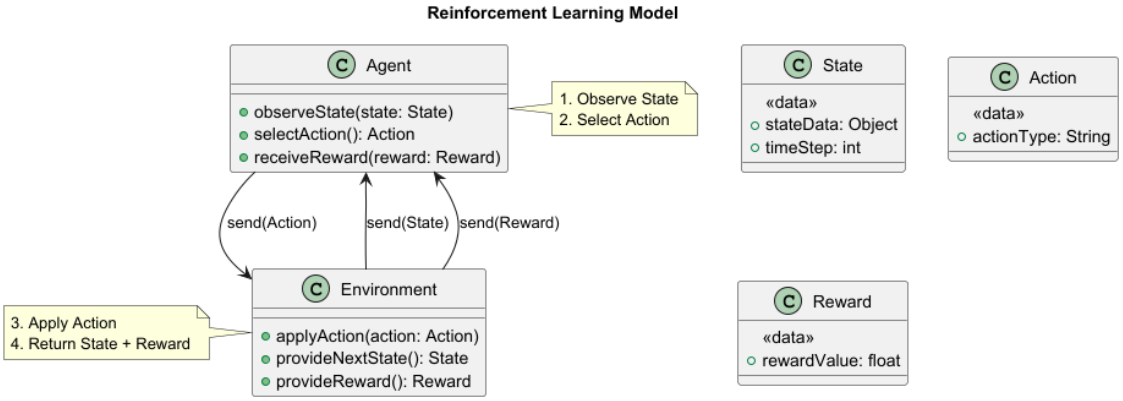


Figure 1 - Reinforcement Learning Model

2.4 Epsilon-Greedy Exploration

The Epsilon-Greedy Exploration [41] strategy is a simple, yet effective approach commonly used in Reinforcement Learning (RL) to balance the trade-off between exploration and exploitation. In this method, an agent chooses actions primarily based on its current knowledge but occasionally explores other options to discover potentially better strategies. This balance is controlled by a parameter, epsilon (ε), which determines the probability of exploration.

Epsilon-Greedy is widely used because of its simplicity and effectiveness in balancing exploration and exploitation. It ensures that an agent does not become overly reliant on current knowledge while still leveraging learned strategies. The method allows the agent to explore new actions with probability ε, while exploiting known optimal actions with probability $1-\varepsilon$. The method requires tuning only one parameter, epsilon, which can be fixed or decay over time. It is also flexible and can adapt to various environments by adjusting epsilon dynamically based on the stage of learning.

In automated software testing, Epsilon-Greedy Exploration can guide the testing process by enabling systematic exploration of software code paths while leveraging previously acquired knowledge about high-risk areas. Epsilon-Greedy can help select between known high-coverage test cases and unexplored paths, improving test coverage. It can also guide the generation of inputs in fuzz testing by prioritizing paths that are both unexplored and potentially vulnerable. Additionally, the strategy can alternate between calling known sequences and experimenting with new API calls to identify overlooked defects.

While Epsilon-Greedy is simple and effective, it has limitations, particularly in environments with complex reward structures. A fixed value of epsilon may lead to suboptimal exploration if not appropriately tuned. High epsilon values can result in unnecessary exploration, wasting resources on less relevant areas. Additionally, Epsilon-Greedy does not inherently account for the complexity of dynamic or highly non-stationary environments.

Epsilon-Greedy Exploration is particularly beneficial for automated software testing due to its simplicity and adaptability. It ensures that testing strategies explore new areas while exploiting known high-coverage paths. By tuning epsilon, the approach can allocate computational resources more efficiently. Its simplicity also makes it easy to integrate into existing automated testing frameworks.

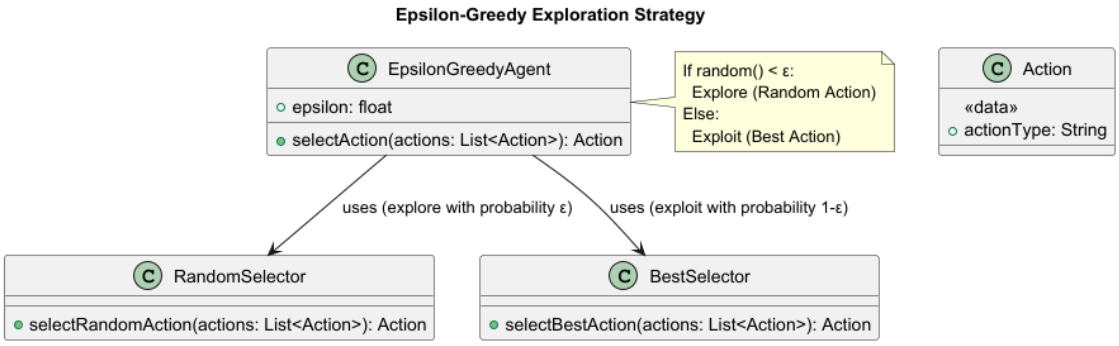


Figure 2 - Epsilon-Greedy Exploration Strategy

2.5 Boltzmann Exploration

Boltzmann Exploration [42] is a more sophisticated exploration strategy in Reinforcement Learning (RL) that uses a probabilistic approach to select actions. Unlike Epsilon-Greedy, which randomly explores actions, Boltzmann Exploration assigns a probability to each action based on its expected reward, favoring actions with higher rewards while still allowing fewer promising actions to be explored.

Boltzmann Exploration provides a more nuanced approach to exploration and exploitation, leveraging temperature parameters to control the randomness of action selection. Actions are selected based on a probability distribution derived from their estimated rewards. A parameter, temperature (T), governs the degree of exploration, with higher values promoting randomness and lower values emphasizing exploitation. Action probabilities are computed using a softmax function, ensuring that all actions have a chance of being selected.

In automated software testing, Boltzmann Exploration is particularly suited for testing scenarios where actions, such as test cases or input generation, vary significantly in their likelihood of uncovering defects. The method can prioritize test cases with higher probabilities of finding defects while still exploring less likely paths. By weighing inputs based on their historical effectiveness, Boltzmann Exploration can generate inputs that are more likely to reveal

vulnerabilities. The approach also adjusts dynamically to changing software states, making it effective for testing stateful systems.

Despite its advantages, Boltzmann Exploration has certain challenges that need to be addressed for effective application. The temperature parameter requires careful tuning to balance exploration and exploitation effectively. Calculating action probabilities using softmax can be computationally intensive for environments with a large action space. Additionally, the strategy may overly prioritize actions with slightly higher rewards, potentially neglecting other promising options.

Boltzmann Exploration offers several advantages for automated software testing, especially in complex and dynamic environments. It focuses on actions with higher potential payoffs, improving testing efficiency. The strategy also adjusts action probabilities in real time, ensuring relevance as the software evolves. By exploring actions probabilistically, it ensures thorough coverage of both high and low-reward paths. The temperature parameter can be adjusted dynamically to suit the testing phase, balancing exploration early on and exploitation later.

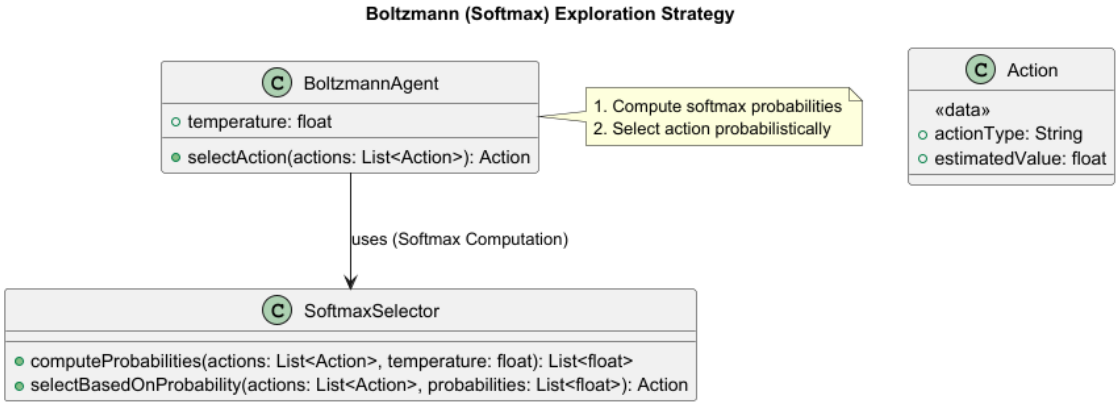


Figure 3 - Boltzmann Exploration Strategy

2.6 Upper Confidence Bound (UCB) Exploration

Upper Confidence Bound (UCB) Exploration [43] is an advanced exploration strategy in Reinforcement Learning that balances exploration and exploitation by explicitly considering both the expected reward of an action and the uncertainty associated with it. Unlike Epsilon-Greedy and Boltzmann Exploration, UCB leverages statistical principles to guide action selection, prioritizing actions with high potential rewards or insufficient exploration.

UCB relies on confidence intervals to make decisions, favoring actions that are either underexplored or have demonstrated high rewards. Its key features include formalizing the exploration-exploitation trade-off by balancing the mean reward of an action with an exploration term that decreases as the action is sampled more frequently. UCB uses statistical confidence bounds to guide exploration, ensuring a principled and efficient exploration process.

The algorithm dynamically shifts focus from exploration to exploitation as more data is gathered about the environment.

UCB's statistical foundation makes it highly effective in scenarios where uncertainty plays a significant role, such as automated software testing. Its ability to focus on high-uncertainty actions can uncover vulnerabilities in rarely tested areas. In automated software testing, UCB helps prioritize test cases that are underexplored but potentially high-impact, ensuring balanced coverage. By identifying inputs that are infrequently tested, UCB can generate new inputs that target unexplored vulnerabilities. UCB is also well-suited for identifying and testing rarely executed code paths that might contain critical defects. In systems with complex state transitions, UCB ensures that states with limited data are explored more thoroughly.

While UCB is a powerful strategy, it has certain limitations that can impact its effectiveness in some testing scenarios. One challenge is the computational complexity of calculating confidence bounds for all actions at each step, which can be computationally intensive, especially in large action spaces. Additionally, the exploration constant (c) requires careful tuning to balance exploration and exploitation effectively. In environments with sparse rewards, UCB may also take longer to converge as it prioritizes underexplored actions.

UCB provides several key advantages for automated software testing, particularly in environments with significant variability or uncertainty. By incorporating uncertainty into action selection, UCB ensures a systematic balance between exploring untested paths and exploiting known high-reward actions. This leads to increased test coverage as UCB naturally focuses on underexplored areas. Furthermore, by prioritizing high-potential actions, UCB optimizes the allocation of computational resources during testing. The approach also adjusts its focus dynamically as more data is gathered, ensuring that testing strategies remain relevant and effective.

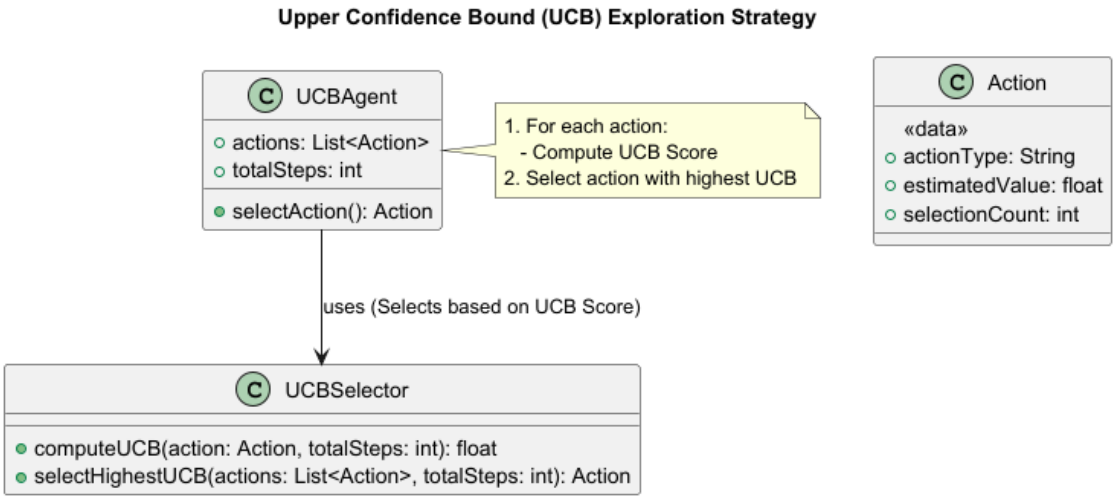


Figure 4 - Upper Confidence Bound Exploration Strategy

2.7 Comparison between Explorations

Epsilon-Greedy, Boltzmann Exploration and Upper Confidence Bound (UCB) Exploration are three widely used strategies in Reinforcement Learning (RL) that address the exploration-exploitation trade-off in different ways. Each approach has its strengths and weaknesses, making them suitable for different scenarios in automated software testing.

Epsilon-Greedy introduces randomness into decision-making by assigning a fixed probability, ϵ, for random exploration. This ensures that even unexplored or low-reward actions are occasionally chosen, preventing the algorithm from prematurely converging on suboptimal solutions. It is simple and effective in smaller environments. Boltzmann Exploration employs a probabilistic approach based on the expected rewards of each action. By using the softmax function, it provides a more balanced exploration, favoring higher-reward actions while still sampling lower-reward actions based on their potential. UCB Exploration leverages statistical confidence intervals to guide decision-making. It selects actions based on both their estimated rewards and the uncertainty of those estimates. This makes UCB particularly effective in environments where systematic exploration is crucial, such as software testing with sparse reward signals. Each strategy's approach to balancing exploration and exploitation has specific implications for automated software testing, where test cases, inputs and paths must be explored effectively to uncover defects.

The strategies differ in how they implement exploration and exploitation, adapt to new information and handle computational requirements. Epsilon-Greedy relies on pure randomness for exploration, ensuring even unlikely options are occasionally tested, but this can result in inefficient exploration. Boltzmann Exploration, by contrast, uses probabilities derived from action rewards, offering a more refined approach. UCB focuses on targeting actions that are underexplored or statistically uncertain, systematically prioritizing them for exploration. In terms of exploitation, Epsilon-Greedy simply selects the best-known action most of the time, whereas Boltzmann Exploration shifts focus toward higher-reward actions using softmax probabilities. UCB takes a calculated approach, exploiting actions with high rewards while accounting for uncertainty, thereby maintaining a balance. Dynamic adaptation varies across the strategies. Epsilon-Greedy offers limited adaptability unless ϵ is decayed over time. Boltzmann Exploration adapts more dynamically by adjusting the temperature parameter, which controls the balance between exploration and exploitation. UCB's dynamic adaptation is inherent in its design, as it continuously evaluates uncertainty and adjusts its exploration priorities accordingly.

Each strategy has specific use cases in automated software testing. Epsilon-Greedy works well in simpler environments, such as initial exploration of software, where test inputs or code paths are relatively small and computational resources are limited. It is also useful for lightweight fuzzing, where random sampling of input combinations is needed. Boltzmann Exploration is more effective in intermediate scenarios like test input prioritization, where inputs with higher rewards (e.g., those that uncover more defects) are favored while still maintaining diversity. It also performs well in fuzz testing, as it balances input diversity with a focus on high-reward

areas. UCB Exploration excels in complex environments, such as stateful system testing, where systematic and uncertainty-aware exploration is essential to discover rare defects or vulnerabilities. By focusing on underexplored actions, UCB ensures comprehensive testing, even in large, dynamic environments.

Each strategy has its limitations. Epsilon-Greedy can be inefficient because its random exploration may waste resources by selecting suboptimal or irrelevant actions. Its fixed ϵ makes dynamic adaptation difficult unless carefully decayed over time. Boltzmann Exploration mitigates some randomness by weighting actions based on expected rewards but requires careful tuning of the temperature parameter. If the temperature is too high, exploration becomes excessive, while if it is too low, it overly favors exploitation. UCB is the most computationally intensive, requiring confidence bounds to be calculated for all actions at each step. This can be challenging in environments with large action spaces or frequent decision points. Additionally, UCB requires sufficient exploration of all actions to generate reliable rewards and uncertainty estimates, which can delay convergence in scenarios with sparse rewards. By understanding these challenges, testers can tailor strategies to their specific testing environments, avoiding pitfalls such as wasted computation or inefficient exploration.

2.8 PRISMA Method

The PRISMA [44] (Preferred Reporting Items for Systematic Reviews and Meta-Analyses) method, first introduced in 1997 and updated in its latest 2020 version, offers a systematic framework for conducting comprehensive literature reviews. This method ensures transparency, objectivity and reproducibility in the review process, making it crucial for answering the research questions of this project. By adhering to the PRISMA guidelines, the review ensures a rigorous approach to literature selection, minimizing bias and enhancing the reliability of the findings.

The PRISMA method begins with defining the main research topic in this case, improving automatic software testing through fuzzing with reinforcement learning and selecting the appropriate research libraries and databases. The databases chosen must provide access to high-quality, peer-reviewed publications, ensuring that only relevant and credible studies are included. Once the research questions are established, they guide the selection of studies. These questions focus on fuzz testing for API vulnerabilities, the role of reinforcement learning in enhancing fuzzing techniques and the limitations of existing fuzzing tools in modern software security testing.

After defining the research questions, key terms are identified for creating search queries. These terms, such as fuzz testing, reinforcement learning, API vulnerabilities and Boltzmann Exploration form the foundation of a targeted search strategy. The search query is constructed to capture a broad set of studies while remaining focused on the key topics of the review. The query is then applied across digital libraries like Google Scholar, IEEE Xplore and ACM Digital Library, with filters to refine results. These filters may include language restrictions (e.g.,

English), document types (e.g., conference papers, technical reports) and specific publication date ranges (e.g., studies published within the past decade) to ensure the relevance and timeliness of the findings.

Once the initial publications are gathered, the PRISMA method proceeds with a rigorous screening process. First, titles and abstracts are reviewed to assess the relevance of each study. Articles that do not align with the research focus or are inaccessible (e.g., behind paywalls or lacking sufficient data) are excluded. Additionally, studies that fall outside the scope such as those unrelated to fuzz testing or reinforcement learning are filtered out.

Following the initial screening, a more detailed full-text review is conducted for the remaining articles. During this phase, the studies are evaluated for their methodological rigor, focusing on the fuzz testing techniques discussed, the integration of reinforcement learning and the proposed solutions to improve vulnerability detection in APIs. Specific attention is given to studies that explore Boltzmann Exploration as an alternative to traditional methods like Epsilon-Greedy. Boltzmann allows for more nuanced exploration-exploitation dynamics, offering more efficient ways to explore API input spaces and improving the detection of high-impact vulnerabilities.

Another key aspect examined is the integration of new metrics like cumulative reward per episode and the exploration vs. exploitation ratio. These metrics help assess the fuzzers effectiveness, not just in detecting vulnerabilities but also in adapting to complex API structures. By providing insights into the fuzzers learning process, these metrics offer a more sophisticated analysis of the tool's efficiency and ability to balance exploration and exploitation.

Once relevant studies are identified, the review process progresses to synthesizing the findings. This synthesis focuses on key trends, common themes and gaps in the current literature. One major limitation identified in existing fuzz testing tools is their inability to effectively handle complex, nested API inputs or their low coverage in terms of vulnerability detection. Many fuzzers are also constrained by their inability to adapt intelligently to the API under test, particularly in environments with dynamic inputs or non-trivial API structures. The integration of reinforcement learning specifically models using Boltzmann Exploration addresses these shortcomings by enabling more adaptive and targeted fuzzing strategies, which can navigate complex API state spaces more effectively.

The review also discusses the application of novel reinforcement learning algorithms tailored to API fuzz testing. These algorithms are designed to explore diverse input combinations intelligently and leverage dynamic reward structures for continuous improvement in detection efficiency. Metrics such as cumulative reward per episode and exploration/exploitation ratios provide vital feedback to enhance the tool’s performance.

Finally, the systematic review identifies research gaps and future exploration opportunities. These include further work on scalable fuzzing tools that incorporate reinforcement learning, the development of real-time fuzzing models for production environments and the

investigation of alternative exploration strategies like Boltzmann Exploration that could improve the robustness of fuzz testing for modern web APIs. By following the PRISMA method, this review aims to offer a transparent, comprehensive overview of the current state of fuzz testing, highlight areas where reinforcement learning and advanced strategies like Boltzmann Exploration could bring improvements and outline opportunities for future research and tool development.

2.9 Systematic Literature Review

This systematic literature review aims to explore the evolving role of reinforcement learning (RL) algorithms in enhancing the effectiveness of fuzz testing techniques, particularly for API security testing. The focus is on comparing traditional exploration strategies like Epsilon-Greedy with more sophisticated approaches, such as Boltzmann Exploration and investigating how these methods can optimize the fuzzing process for detecting software vulnerabilities.

This review investigates the integration of Reinforcement Learning into fuzz testing, specifically focusing on two main strategies for exploration, Epsilon-Greedy and Boltzmann Exploration. Epsilon-Greedy is a popular technique for balancing exploration and exploitation in RL, but it often struggles with fine-tuning the exploration of complex input spaces. Boltzmann Exploration, on the other hand, offers a probabilistic approach, providing more nuanced control over exploration-exploitation dynamics, which can improve the efficiency and adaptability of fuzz testing for complex API structures.

In addition to the exploration strategies, this review also examines the new metrics introduced for evaluating fuzz testing performance, such as the cumulative reward per episode and exploration/exploitation ratios. These metrics help provide a more refined assessment of a fuzzers ability to detect vulnerabilities and improve its adaptability to different API environments. To gather relevant data, the Publish or Perish software tool [45], utilizing Google Scholar as the primary source, was employed. This tool allows for efficient extraction of academic articles and publications, facilitating a comprehensive search for scholarly works.

The query constructed was:

Reinforcement Learning AND (exploration strategies OR Boltzmann OR Epsilon-Greedy) AND (fuzzing metrics OR cumulative reward OR exploration vs exploitation ratio)

This query was designed to capture studies exploring the intersection of fuzz testing, reinforcement learning, exploration strategies (like Epsilon-Greedy and Boltzmann Exploration) and the performance metrics used to evaluate these approaches in API security testing. By incorporating these keywords, the query aims to ensure a comprehensive review of the latest studies on RL-enhanced fuzz testing techniques and the application of these methods in automated security testing for modern APIs. Following the construction of the search query, inclusion and exclusion criteria were defined to filter the results, ensuring that only the most relevant and high-quality studies were considered. These criteria are summarized as follows:

Inclusion Criteria Number Description	Description
IC1	The paper must have been published between 2019 and 2024.
IC2	The paper must be written in English.
IC3	The paper should be relevant to informatics, technology or IT disciplines.

Table 1 - Systematic Review Inclusion Criteria

Exclusion Criteria Number Description	Description
EC1	Papers that do not focus on Reinforcement Learning techniques (like Boltzmann Exploration, Epsilon-Greedy or other RL-based strategies) in the context of fuzz testing or API security.
EC2	Papers that do not discuss API vulnerabilities, API security testing or automated fuzz testing for APIs.
EC3	Papers that primarily discuss older or obsolete fuzzing techniques (e.g., traditional black-box testing methods) and do not consider modern advancements in fuzzing, such as machine learning and reinforcement learning integration.

Table 2 - Systematic Review Exclusion Criteria

This review synthesizes the findings of studies that integrate Reinforcement Learning techniques, particularly focusing on exploration strategies such as Epsilon-Greedy and Boltzmann Exploration and examines how these strategies contribute to the efficiency and effectiveness of fuzz testing. The review highlights the advantages of Boltzmann Exploration over Epsilon-Greedy in optimizing fuzz testing for complex API structures, providing a more balanced exploration-exploitation mechanism that can significantly enhance vulnerability discovery. Additionally, this review examines the new performance metrics introduced to assess fuzzing effectiveness, including the cumulative reward per episode and the exploration/exploitation ratios. These metrics help to evaluate not just the detection of vulnerabilities but also the adaptability of the fuzzer to various API environments. The review also explores the challenges faced by researchers and practitioners in applying fuzz testing to real-world API security scenarios, such as the scalability of fuzzing techniques, the generation of high-quality test inputs and the trade-offs between coverage and performance.

Finally, the review identifies gaps in the existing research, including the need for scalable fuzz testing tools that leverage reinforcement learning algorithms, more sophisticated exploration

strategies and the development of real-time fuzzing models for production environments. By synthesizing the findings of selected studies, this review provides a comprehensive understanding of the current state of fuzz testing and its enhancement through reinforcement learning, highlighting future research opportunities to improve automated security testing for APIs.

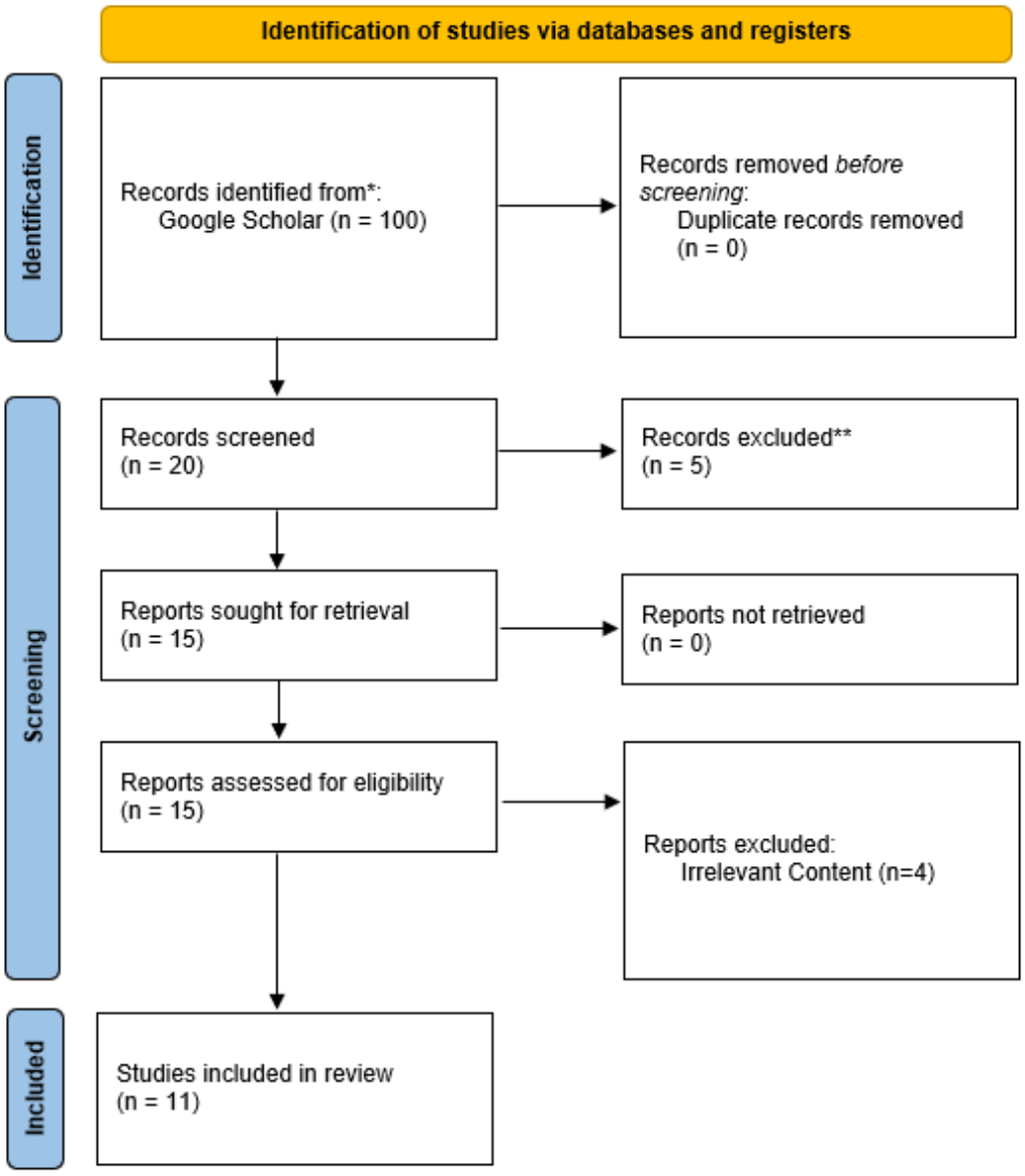


Figure 5 - PRISMA Diagram of Fuzzer API Testing Systematic Review

2.10 Collected Studies

After applying the inclusion and exclusion criteria, articles addressing technologies, methodologies and challenges associated with fuzz testing, reinforcement learning Boltzmann and Epsilon-Greedy explorations and metrics for cumulative reward and exploitation vs exploration ratio were selected. These studies have been organized into a table, classified according to the research questions they address.

Research Question	Relevant Studies	Answer/Insight
RQ1: What are the limitations and flaws in the current generation of RESTful API fuzzers that use Reinforcement Learning for input generation?	Amin, MRRM, & Othman, MF (2021) - Re-exploration of ε-greedy in deep reinforcement learning [46]	Limitation: The ε-greedy algorithm suffers from suboptimal exploration in certain complex systems like APIs. This can lead to missing critical vulnerabilities due to an over-exploitation of certain paths while not fully exploring others.
	D'Eramo, C, Cini, A, & Restelli, M (2019) - Exploiting action-value uncertainty to drive exploration in reinforcement learning [47]	Limitation: Traditional RL methods, such as ε-greedy, fail to properly handle uncertainty in API input spaces, potentially resulting in ineffective fuzzing that misses deep or hard-to-reach vulnerabilities.
	Zangirolami, V, & Borrotti, M (2024) - Dealing with uncertainty: Balancing exploration and exploitation in deep recurrent reinforcement learning [48]	Limitation: There is a significant gap in handling the exploration-exploitation trade-off in complex systems like RESTful APIs. A lack of balance could lead to inadequate vulnerability discovery, especially when encountering unknown or poorly understood API structures.
RQ2: What additional metrics and techniques can be explored to improve the analysis and testing capabilities of RESTful API fuzzers?	Sharma, K, Singh, B, Herman, E, (2021) - Maximum information measure policies in reinforcement learning with deep energy-based model [49]	Insight: Integrating information-theoretic metrics such as mutual information into the exploration process can improve the fuzzers' ability to target vulnerabilities more efficiently by focusing on areas of the API with higher information gain.

	Zangirolami, V, & Borrotti, M (2024) - Dealing with uncertainty: Balancing exploration and exploitation in deep recurrent reinforcement learning [48]	Insight: Techniques for managing uncertainty in RL models (e.g., using recurrent networks or uncertainty-based exploration) can be adapted for fuzzing to improve the robustness and efficiency of vulnerability discovery in APIs.
	Sukhija, B, Coros, S, Krause, A, Abbeel, P (2024) - Boosting exploration in reinforcement learning through information gain maximization [50]	Insight: By incorporating information gain maximization into the fuzzing process, fuzzers can prioritize input generation that maximizes the exploration of API vulnerabilities, leading to better testing coverage.
RQ3: How do different exploration strategies, including Boltzmann Exploration, compare to traditional algorithms (such as Epsilon-Greedy) in terms of their effectiveness in generating diverse and targeted inputs for discovering vulnerabilities in RESTful APIs?	Thadikamalla, S, & Joshi, P (2023) - Exploration Strategies in Adaptive Traffic Signal Control [51]	Insight: Boltzmann exploration could outperform ε-greedy by providing more diversified exploration, leading to a broader search of the API's input space and potentially uncovering previously missed vulnerabilities.
	Gupta, H, Kong, ST, Srikant, R, & Wang, W (2019) - Almost Boltzmann exploration [52]	Insight: Boltzmann exploration offers more controlled exploration compared to ε-greedy by adjusting exploration based on the probability distribution of actions, leading to more targeted fuzzing and potentially more effective vulnerability discovery.
	Zhang, S, Li, S, Wu, F, & Li, XY (2023) - Reinforcement Learning for Node Selection in Mixed Integer Programming [53]	Insight: Strategies like Boltzmann exploration can adaptively select more promising nodes (or inputs), which can improve the exploration of API structures and better identify security vulnerabilities compared to traditional algorithms.

RQ4: How can the exploitation-exploration balance and reward-based learning metrics, such as the exploitation vs. exploration ratio and cumulative reward per episode, be used to enhance the exploration of API structures and improve vulnerability detection during API testing?	Shani, L, Efroni, Y, & Mannor, S (2019) - Exploration conscious reinforcement learning revisited [54]	Insight: By carefully tuning the exploitation-exploration balance and monitoring the cumulative reward per episode, fuzzers can focus their efforts on the most vulnerable API areas, improving the overall effectiveness of vulnerability detection.
	Tiwari, PK, Singh, P, Rajagopal, NK (2023) - IoT-Based Reinforcement Learning Using Probabilistic Model for Determining Extensive Exploration [55]	Insight: A probabilistic model that adjusts exploration based on reward accumulation can be used in fuzzing to dynamically balance exploration and exploitation, enhancing the discovery of hidden or rare vulnerabilities in APIs.
	Li, M, Huang, T, & Zhu, W (2021) - Adaptive exploration policy for exploration–exploitation tradeoff in continuous action control optimization [56]	Insight: Reward-based metrics such as cumulative reward per episode can optimize the exploration-exploitation trade-off, allowing fuzzing algorithms to focus on high-reward (vulnerable) parts of the API, improving testing effectiveness.

Table 3 - Collected Studies

With the collected studies, the research questions can be answered in a comprehensive and detailed manner by leveraging the insights and findings from a wide range of approaches in fuzz testing, machine learning and reinforcement learning. These studies provide valuable information on the limitations of current techniques, such as scalability issues and inefficiencies in vulnerability detection, while also offering potential solutions for improving fuzz testing methods. By exploring the effectiveness of various exploration strategies like Epsilon-Greedy and Boltzmann Exploration, as well as the evaluation of new metrics for testing API vulnerabilities, the research sheds light on novel strategies for enhancing the fuzzing process.

RQ1 regarding the limitations and flaws in current RESTful API fuzzers using reinforcement learning is addressed by studies focusing on exploration-exploitation balances, inefficiencies in generating diverse inputs and the overall effectiveness of existing fuzzing tools in addressing complex API vulnerabilities.

RQ2, which investigates additional metrics and techniques to improve testing capabilities, is informed by studies that propose enhancements to the fuzzing process, including dynamic exploration policies, multi-attribute decision-making models and the use of more adaptive exploration strategies to increase the accuracy and efficiency of fuzzing tools.

For RQ3, the comparison of different exploration strategies, such as Boltzmann Exploration versus Epsilon-Greedy, reveals differences in their effectiveness. Some studies indicate that Boltzmann Exploration, by prioritizing exploration based on a probabilistic distribution, offers more diversity in input generation, which can help identify previously overlooked vulnerabilities in APIs.

Finally, RQ4 on the exploitation-exploration balance and the use of reward-based learning metrics for enhancing API vulnerability detection is explored through various studies focusing on the cumulative reward per episode and the exploitation vs. exploration ratio, revealing that careful balancing of these metrics can significantly improve vulnerability detection during API testing. These insights provide a deeper understanding of how reinforcement learning models, particularly when combined with dynamic exploration strategies, can advance the state of automated security testing for APIs.

This comprehensive approach, combining the findings of the studies, presents new opportunities to refine current fuzz testing methodologies and enhance the efficiency of vulnerability detection in RESTful APIs, particularly through the integration of more sophisticated reinforcement learning techniques and exploration strategies.

3 Existing System Analysis

The existing system under analysis, FuzzTheRest, is an automated software testing framework specifically tailored for fuzzing RESTful APIs using reinforcement learning (RL) algorithms. The system's design is rooted in modularity and scalability, ensuring it can handle a variety of API structures while optimizing the testing process through intelligent learning mechanisms. This system incorporates three core components: FuzzCore, the RL-based Fuzz Algorithm and a User Interface (UI). Each of these components is built to handle specific responsibilities, yet they work in harmony to deliver efficient, automated and adaptive API security testing. Below is a detailed breakdown of the architecture and the role each component plays.

3.1 Overview of System Architecture

The architecture of the system is designed to support continuous learning, test generation and visualization results. It is divided into three primary layers:

- **Fuzz Core**, which serves as the backbone of the testing system, handling core logic and backend operations such as API parsing, input injection, test execution and data aggregation.
- **Fuzz Algorithm**, which integrates advanced reinforcement learning techniques to guide the generation of inputs. This component is responsible for balancing exploration (trying new inputs) and exploitation (leveraging past knowledge) to discover hidden vulnerabilities effectively.
- **User Interface (UI)**, which functions as the front-end layer that enables user interaction with the system. It allows users to configure test settings, view real-time feedback, interpret testing metrics and generate comprehensive reports.

This tri-layered architecture ensures the system is both technically robust and user-friendly. The components communicate through well-defined APIs and data structures, which allows for flexibility in upgrading specific modules or integrating with other tools.

3.2 Fuzz Core

At the middle of the FuzzTheRest system lies the Fuzz Core, which serves as the backbone for all backend logic, input processing and test execution. The Fuzz Core is engineered to provide modular, extensible architecture that can support advanced fuzzing strategies while remaining flexible enough for integration with external tools and interfaces. It enables the seamless execution of fuzzing campaigns by orchestrating the flow of data from API specification parsing through to test execution and result aggregation.

One of the primary responsibilities of the Fuzz Core is offering an API endpoint that accepts file paths and associated identifiers as input and processes using the parser_service. The parsing process involves converting raw definitions into structured internal representations, which are then further transformed into domain-specific HTTP models by the mapper_service. These representations include critical components such as HTTP requests, parameters and request bodies, all modeled in a way that supports seamless integration with the fuzzing logic. This plays a pivotal role in managing the lifecycle of fuzzing operations. It acts as a conduit between the front-end user interface and the core backend services, accepting commands to start, stop or monitor fuzzing campaigns. Behind the scenes, it relies on the orchestration service, a key component responsible for executing and coordinating individual fuzzing jobs. This service ensures that all tasks are initialized correctly, executed efficiently and terminated cleanly upon completion, maintaining robust control over the testing environment. The mapper service is essential for the transformation of parsed API data into actionable models. It serves as a bridge between the parsing logic and the fuzzing environment by converting data into objects such as HTTP Request, Request Body and Parameter. These representations ensure that each API element is modeled consistently, supporting accurate test generation and analysis.

Data persistence and analytics are handled by the MongoDB Service, which stores vital fuzzing data including test results, metrics, crash logs and configuration settings. This database layer allows for efficient querying and retrieval of past results, aiding in debugging, reporting and long-term analysis. Meanwhile, the parser service continues to be the core utility for extracting relevant information from the files. It meticulously processes schemas, parameter definitions and endpoint structures, feeding this information into the system in a structured and standardized manner.

In terms of workflow, the system follows a clear, logical path:

1. The fuzz core controller receives the request.
2. The file is parsed by the parser service and transformed into internal representations by the mapper service.
3. Fuzzing tasks are initiated through the orchestrator controller, which passes control to the orchestration service.
4. Tests are executed and all results, including crash logs and metrics, are stored using the MongoDB service.
5. Data models defined with the defined taxonomy ensure consistent and structured data management throughout the process.

Overall, the Fuzz Core's layered and modular design supports robust, scalable and intelligent fuzz testing, capable of adapting to diverse API structures while supporting advanced reinforcement learning algorithms on the backend.

3.3 Fuzz Algorithm

The fuzzing mechanism within the system is driven by a reinforcement learning-based architecture, specifically a Q-learning algorithm orchestrated through a collection of modular services and components. The overall workflow begins with the fuzzing Controller, which acts as the central coordinator of the fuzzing process. Built using FastAPI, this controller handles incoming HTTP requests and dynamically determines which algorithm type and configuration should be applied.

When a fuzzing request is submitted through the /execution endpoint, the controller invokes the Parsing Service, which is responsible for processing and structuring raw input data. It parses key components such as attributes, objects, schemas and request bodies, converting them into domain-specific classes like Attribute, Object, Schema and HTTP Request. This transformation ensures that the fuzzing logic operates on well-defined and consistent representations of the API structure. Once the input has been parsed and validated, the controller uses the create_algorithm_data function to identify and configure the correct fuzzing strategy. In scenarios where Q-learning is selected, the execution is passed to the QlearningService, which contains the core logic of the reinforcement learning-based fuzzer.

The QlearningService initializes the APIFuzzyTestingEnvironment, a simulated environment that facilitates safe interaction with the API under test. Within this environment, a QLearningAgent is deployed, which approaches the fuzzing task as a Markov Decision Process (MDP) where each API interaction represents a transition between states. A critical aspect of the fuzzing algorithm is action selection, which determines which test inputs the agent will attempt. To manage the balance between exploration (trying new, potentially useful inputs) and exploitation (reusing inputs that previously led to errors), the algorithm employs the Epsilon-Greedy exploration strategy. This Epsilon-Greedy mechanism ensures that the algorithm does not get stuck in local optima by periodically introducing randomness into its decisions. As learning progresses, epsilon may decay gradually, shifting the balance from exploration toward exploitation to fine-tune input sequences that are more likely to cause unexpected behavior.

Each test input is evaluated and a reward is computed based on the observed outcome. The system gives positive rewards for responses that indicate abnormal or critical behavior, such as server errors (e.g., HTTP 500), exceptions or application crashes. Neutral or negative rewards are assigned for typical, uneventful responses. This reward feedback is used to update the Q-values associated with specific input sequences, thereby guiding the agent to prefer more promising paths in future iterations.

The APIFuzzyTestingEnvironment component simulates the entire lifecycle of interaction with the target API, handling request mutations, response parsing, state tracking and reward calculation. It provides a structured and isolated testbed where the RL agent can experiment with various inputs while monitoring key performance and safety indicators. After the fuzzing session concludes, the system aggregates all relevant information including reward statistics, execution duration, crashes and coverage metrics and sends the structured results back to the controller. The controller then responds to the original API request, delivering a complete report to the user.

In summary, this fuzzing algorithm represents an intelligent and adaptable system for testing RESTful APIs. By integrating reinforcement learning and specifically leveraging Epsilon-Greedy exploration, the system can systematically uncover vulnerabilities that traditional fuzzers based on random or brute-force methods might overlook. Over time, the agent learns to optimize its input selection process, enabling more effective and efficient exploration of complex API surfaces.

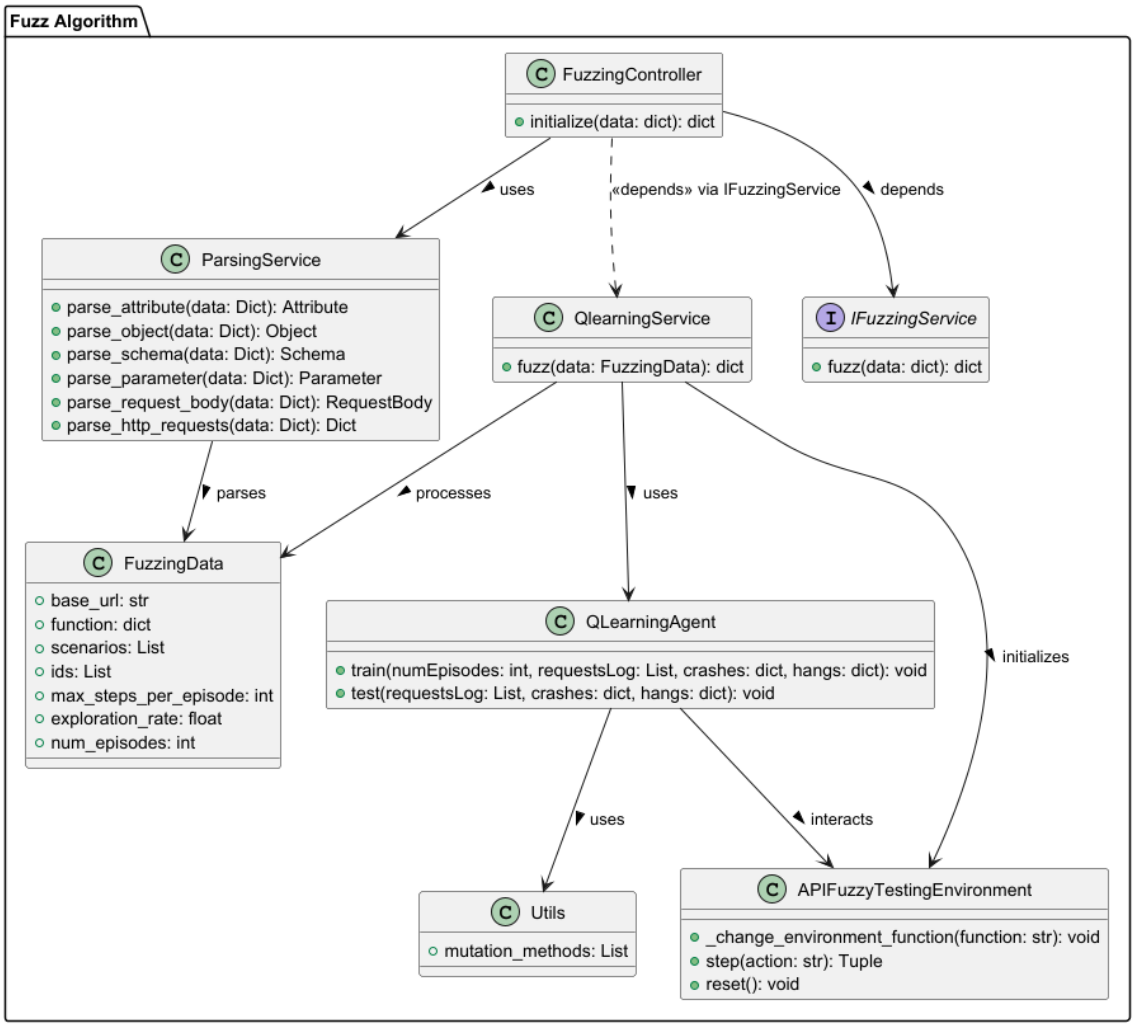


Figure 6 - Fuzz Algorithm Class Diagram

3.4 Existing System Agent Workflow

To further illustrate the interaction between the components described above, the following sequence diagram presents a detailed visualization of the dynamic behavior of the existing FuzzTheRest system during a typical fuzzing campaign. The diagram depicts the flow of actions beginning with the initialization of the QLearningAgent and its coordination with the APIFuzzyTestingEnvironment, which provides the structured and isolated environment necessary for safe API interaction. The agent proceeds through iterative cycles of choosing actions based on a reinforcement learning policy, applying mutation methods, sending mutated requests to the API server and receiving and processing responses. In cases of API anomalies, such as crashes or timeouts, the system engages specialized crash and hang tracking modules, ensuring that critical failures are properly logged and analyzed.

Throughout the fuzzing process, the agent dynamically updates its Q-values based on received rewards, refining its input generation strategy to prioritize high-value mutations. At the conclusion of each episode, cumulative metrics, including rewards, execution durations and API state transitions, are collected and stored. Once the training or testing cycle is complete, the system compiles the gathered data into structured reports, generating visualizations such as cumulative reward graphs, exploration-exploitation ratio charts, Q-value convergence plots and action distribution diagrams, which provide comprehensive insight into the fuzzing session’s performance. The architecture revealed by the sequence diagram emphasizes the separation of concerns between the core environment simulation, the learning mechanism and the data analysis layers. It shows how each component interacts seamlessly through defined interfaces, ensuring modularity, extensibility and ease of maintenance.

This following visualization captures the orchestration between training, exploration, API interaction, exception handling, metric aggregation and reporting and provides a clear structural reference for understanding the current system's operational dynamics and for identifying future areas of improvement.

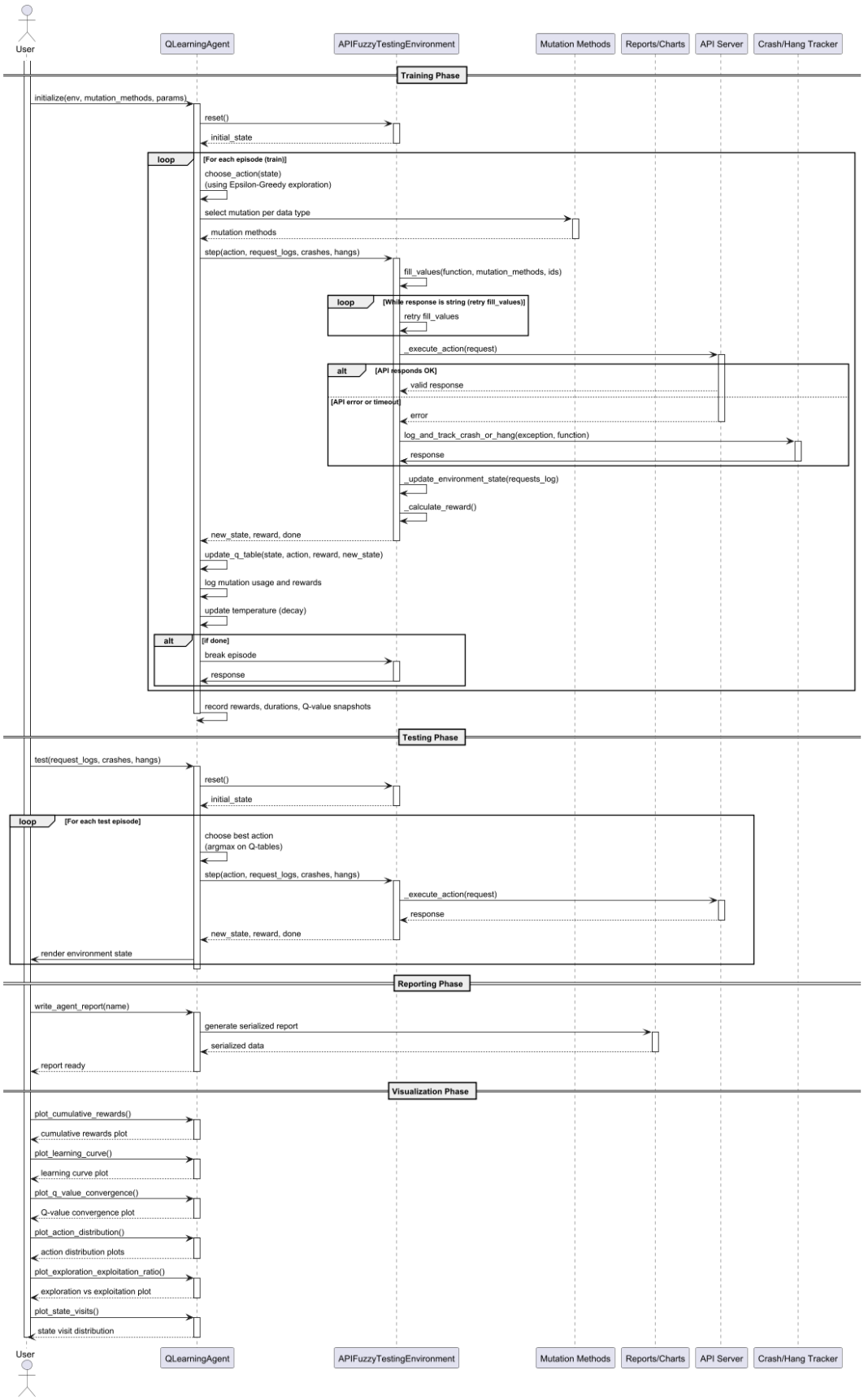


Figure 7 - Existing System Agent Workflow

4 Design of the solution

The proposed solution integrates reinforcement learning with a Q-learning agent tailored to guide fuzzing strategies for testing API endpoints. The agent operates in a custom environment designed to simulate interactions with APIs through input mutation, reward feedback and state transitions. Initially implemented with ε-greedy exploration, the agent was later enhanced by incorporating Boltzmann (Softmax) exploration, which provides a more probabilistic and adaptive method for balancing exploration and exploitation during learning [57].

4.1 Action Selection via Boltzmann Exploration

A key enhancement in the agent's design is the replacement of the ε-greedy strategy with Boltzmann (Softmax) exploration, which enables more nuanced and probabilistic action selection. In the updated action method, the agent computes a probability distribution over possible actions by applying the softmax function to the Q-values for each input type. This allows actions with higher expected rewards to be selected more frequently, while still giving lower-valued actions a chance to be chosen, thus maintaining exploration.

The updated method collects Q-values from five separate Q-tables corresponding to integer, float, boolean, byte and string inputs and applies the softmax transformation using a temperature-controlled formula. The temperature parameter determines how sharply the probability distribution favors higher-valued actions. When the temperature is high, the probabilities become more uniform, encouraging exploration. As the temperature decreases, the agent increasingly focuses on exploiting the best-known actions. To support this dynamic, the temperature decays slightly after each episode, promoting a gradual transition from exploration to exploitation. Each training episode begins with the environment resetting to an initial state. The agent then selects one mutation action per input type using the softmax-based action selection strategy [58].

These selected mutations are applied to the input data and sent to the target API. The response from the API is used to calculate a reward, which reflects the impact of the mutation such as whether it triggered a crash, error or unexpected behavior. The Q-tables are then updated using the standard Q-learning update rule, which adjusts the value estimates based on the received reward and the expected future rewards. After this update, the exploration temperature decayed slightly to refine the agent's focus over time. During training, the agent also logs several metrics to support evaluation and debugging. These include the frequency of each mutation method's use, the cumulative rewards associated with each mutation type, the number of times each HTTP status code (representing environment states) is encountered and the duration and reward history of each episode.

To evaluate learning progress, the agent generates a series of plots that offer insight into various aspects of its behavior. Q-value convergence plots track how the average Q-values change over time, indicating whether learning is stabilizing. A learning curve, based on the moving average of rewards, shows how the agent's performance improves across episodes. Cumulative rewards illustrate the overall trend in performance, while action distribution plots highlight which mutation types are most frequently selected. Additionally, exploration versus exploitation behavior is visualized through temperature-based decision trends and state visit frequency plots reveal how thoroughly the agent has explored the space of possible API responses. Following the training phase, the agent undergoes deterministic testing to assess its ability to generalize its learned strategy. During testing, it selects the highest-Q action for each input type, fully exploiting its accumulated knowledge. These actions are applied to the environment and the agent interacts with the API without any randomization. The resulting responses are analyzed to determine how effectively the trained agent can consistently uncover system crashes, hangs or other edge cases. This phase validates the usefulness of the learned policy and the effectiveness of the Boltzmann-based exploration mechanism in guiding the agent toward impactful mutations [59].

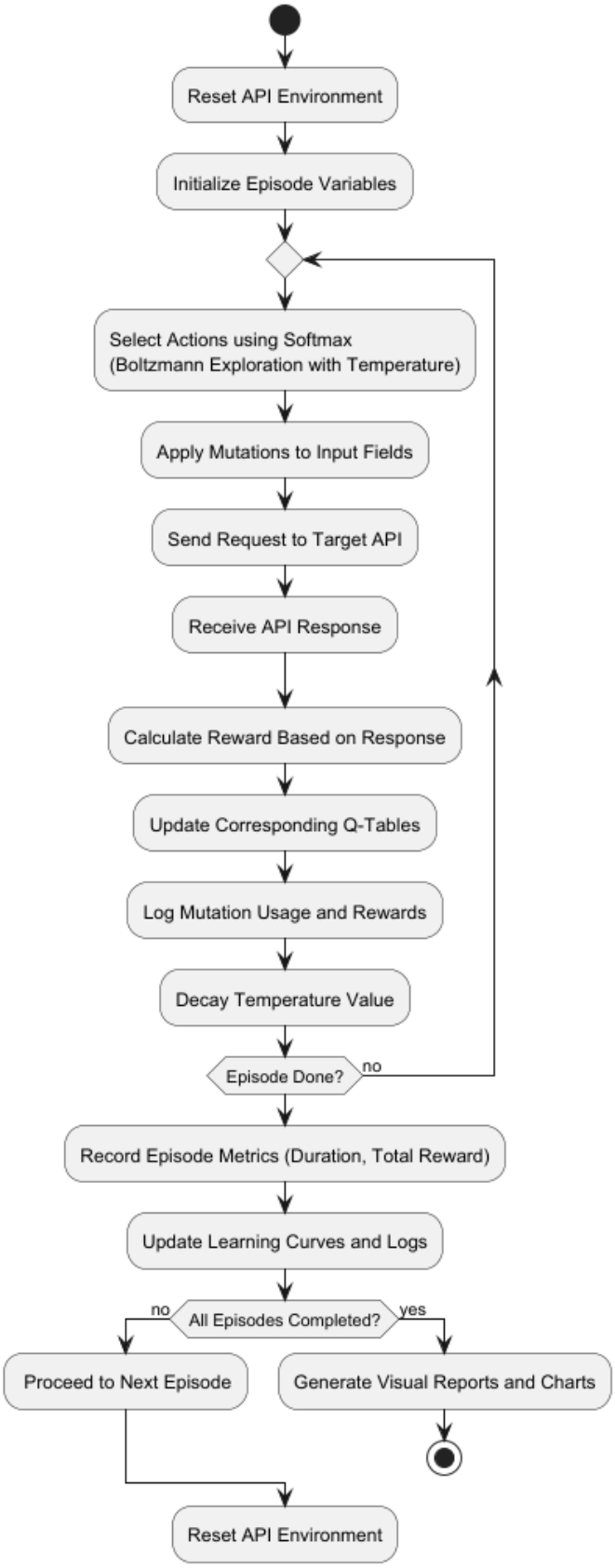


Figure 8 - Flowchart Boltzmann Algorithm

4.2 Boltzmann Exploration Environment Design

In addition to redesigning the action selection process with Boltzmann exploration, the agent's architecture was extended to explicitly manage the temperature dynamics influencing the exploration-exploitation trade-off. A dedicated temperature control module was introduced into the training loop to ensure a structured and gradual transition from exploratory to exploitative behavior over the course of episodes. During each training episode, after the agent selects actions using softmax-transformed Q-values, the current temperature is used to modulate the distribution of action probabilities. This ensures that actions associated with both high and moderate rewards remain accessible early in training, enabling the agent to explore a broader set of behaviors. As episodes progress, the temperature decreases slightly using a predefined decay rate. This decay mechanism is critical, as it systematically reduces randomness, guiding the agent toward a policy that favors high-value actions increasingly over time.

The interaction between the agent, the temperature control module and the environment follows a structured sequence. Initially, the agent queries the Boltzmann module to compute exploration probabilities for each input type based on current Q-values and temperature. After executing the selected mutations and observing the API's responses, the agent updates the Q-tables and requests the Boltzmann module to decay the temperature slightly before the next episode begins. This decoupled design between action selection and temperature management improves the flexibility of the agent, allowing dynamic adaptation of exploration strategies without entangling the core learning logic [60].

The training process is monitored through several visualization tools, including a dedicated Exploration-Exploitation Ratio chart that plots the normalized exploration and exploitation rates across episodes. This visualization provides an immediate overview of the agent's behavioral shift during training, allowing for a detailed assessment of whether exploration was sufficient and whether exploitation was appropriately emphasized during later episodes. Additional plots such as cumulative rewards, learning curves, Q-value convergence and mutation action distribution further support comprehensive analysis. This modular and explicitly controlled temperature design ensures that the Boltzmann-enhanced Q-learning agent can intelligently navigate the fuzzing space, initially favor discovery and progressively specialize in exploiting the most impactful mutation strategies.

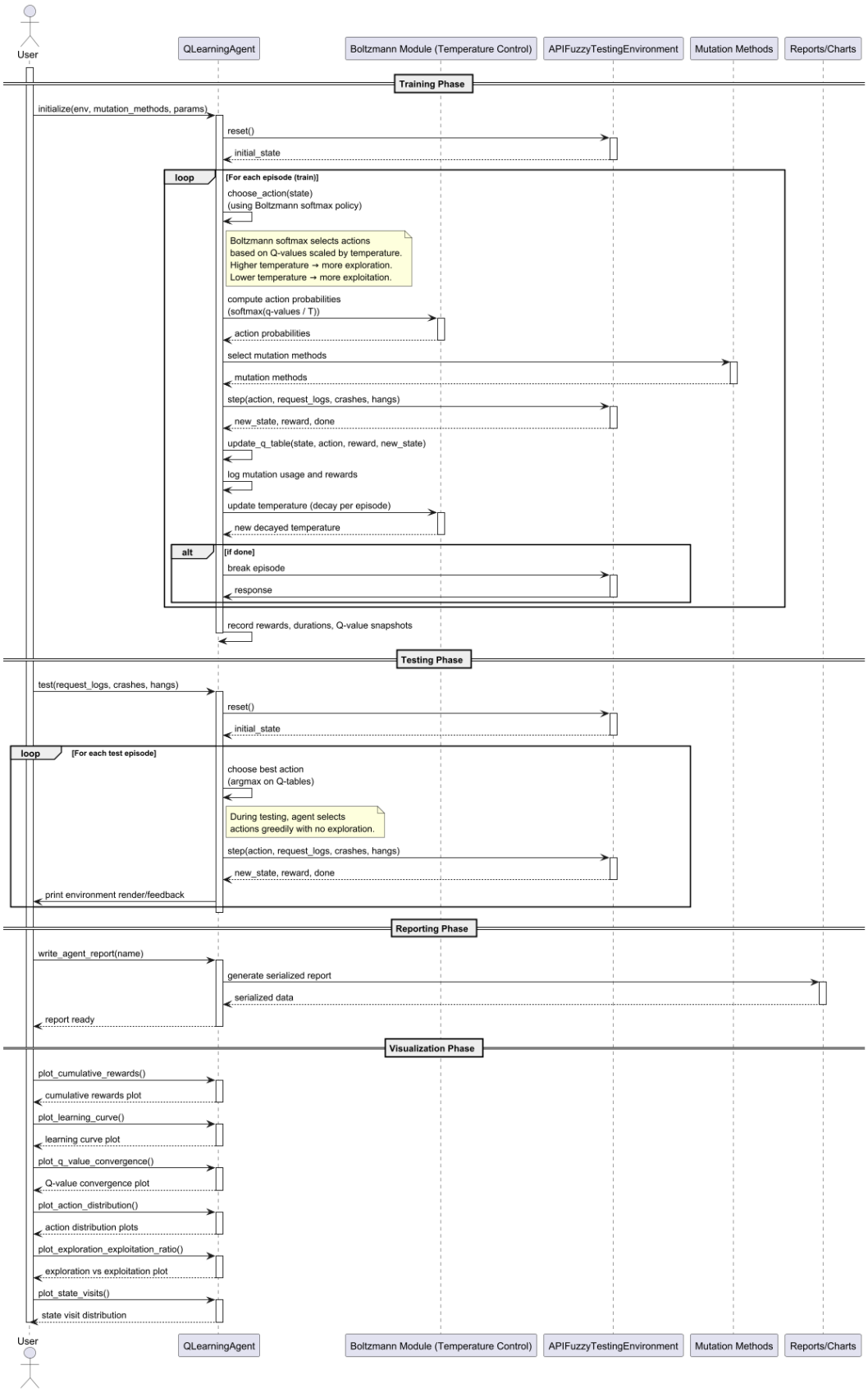


Figure 9 - Boltzmann Exploration and Temperature Control Design

4.3 Flow of Boltzmann Exploration and Temperature Dynamics

To better illustrate the internal mechanics of the enhanced action selection process, a detailed view of the Boltzmann exploration and temperature control flow within the QLearningAgent has been developed. This flow describes how the agent selects actions probabilistically based on learned Q-values, manages temperature decay over episodes and updates its internal state based on environmental feedback [61].

The process begins with the agent retrieving the current Q-values from its five specialized Q-tables (corresponding to integer, float, boolean, byte and string input types). These Q-values represent the estimated expected rewards for each available mutation method. The agent then forwards these Q-values to the Boltzmann Exploration Module, which applies to the softmax transformation using the current temperature parameter to generate a probability distribution over the available actions for each input type. Using these computed probabilities, the agent probabilistically selects one mutation method per input type. This action selection mechanism ensures that higher-valued mutations are more likely to be chosen while maintaining a nonzero probability of exploring fewer promising options, depending on the current temperature setting. Once the actions are selected, the agent applies the corresponding mutations to the API input data and sends the resulting request to the environment (API under test).

The environment processes the mutated request and returns a response, which is evaluated to generate a reward signal reflecting the success or impact of the mutation (e.g., whether it triggered a crash, unexpected behavior or other anomalies). Using this reward feedback, the agent updates its Q-tables by adjusting the value associated with each selected action based on the observed outcome and the estimated future rewards. Following the completion of each episode, the agent requests the Temperature Controller module to decay the current temperature slightly according to a predefined decay rate. This ensures a gradual transition from a highly exploratory behavior in the early episodes to a more exploitative and deterministic behavior as training progresses. The temperature decay mechanism is critical to avoiding premature convergence while still enabling the agent to eventually specialize in the most effective mutation strategies.

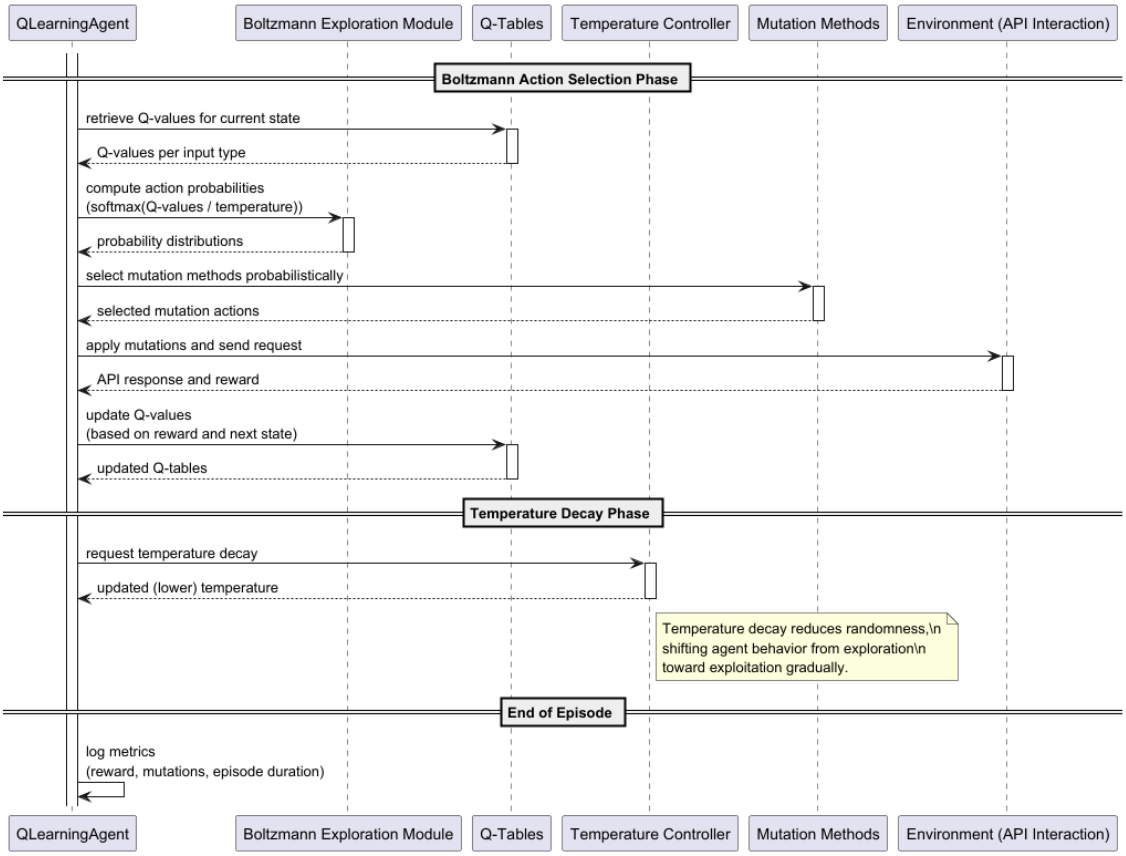


Figure 10 - Flow of Boltzmann Exploration and Temperature Dynamics

This detailed flow is visualized in the dedicated sequence diagram, which captures the interaction between the agent, the Boltzmann module, the temperature controller, the Q-tables, the mutation methods and the API environment. By modularizing these components, the agent's architecture achieves greater flexibility, allowing easy adaptation of exploration strategies without disrupting the core fuzzing workflow. Overall, this detailed control over exploration, action selection and temperature management is pivotal in ensuring that the agent can effectively navigate the large and complex space of possible API inputs, maximizing the discovery of vulnerabilities in a structured and adaptive manner.

5 Implementation of the solution

The implementation of the solution involves several key components from changes in the Q-Learning Agent, environment interactions, action selection via Boltzmann exploration, Q-table updates and the tracking and visualization of learning metrics. Additionally, the solution incorporates a dynamic temperature decay mechanism to facilitate the transition from exploration to exploitation. Furthermore, the system maintains detailed logs of mutation method usage and rewards to enable deeper insights into the agent's learning process. Below is an outline of the key steps in the implementation of the solution.

5.1 Suitable APIs for Testing

A critical aspect of the solution is selecting APIs that represent diverse real-world challenges for the agent to learn effective fuzzing strategies. APIs were selected based on criteria such as data richness, structural complexity, schema variety, authentication methods and domain relevance. This variety ensures that the system is exposed to a broad spectrum of potential edge cases and interaction patterns.

API	Domain	Advantages for Fuzz Testing
Regulations.gov API (GSA) [62]	U.S. Government regulations	Offers rich, nested JSON structures with many distinct endpoints such as dockets, documents and comments. The API contains real-world metadata formats that provide a realistic testing environment.

Dados.gov.pt [63]	Portuguese government open data	This API includes multilingual responses and a wide range of datasets across various sectors. It supports multiple data formats like JSON, CSV and XML, making it useful for testing diverse data handling capabilities.
Data USA API [64]	U.S. demographic and economic statistics	Provides deeply nested, highly structured data with complex queries involving multiple parameters. The response lengths vary significantly, which tests the agent's ability to handle different data sizes.
OpenFDA API [65]	Drug safety and public health	The API has a large dataset with deep filtering logic and real-world regulatory structure. This makes it suitable for testing complex query combinations and identifying edge cases in health data.
U.S. Treasury Fiscal Data API [66]	Federal finance, debt, spending	This API offers time-series and tabular data with a stable and consistent schema. It features versioned endpoints, useful for testing temporal data handling and schema evolution.
NASA Open APIs [67]	Science & Space	Multiple APIs at the disposal, with access to all after requesting the singular API Key. High variability in response content (imagery, telemetry, dates), offers fuzzing of spatial, temporal and scientific metadata.
GitHub REST [68]	Version Control	Multi-entity APIs with deeply nested permissions and tokens. Ideal for permission fuzzing and repo mutation logic.

GitLab API [69]	DevOps & Git	Rich interaction surfaces (issues, pipelines, hooks). Supports deep fuzzing of access control, webhook inputs and CI definitions.
Jenkins API [70]	CI/CD	Job configurations, environment variables and plugin chains introduce high mutation opportunities in execution flows.
Docker Hub API [71]	Containers	Image tags, search filters and large metadata objects allow tag and registry fuzzing, useful for injection and cache test cases.
Kubernetes API [72]	Orchestration	Rich resource schemas, lifecycle operations and RBAC support make it suitable for state transition, privilege escalation and CRD fuzzing.
Terraform Cloud API [73]	Infrastructure as Code	Highly structured stateful data, plan/apply operations and resource graphs make this ideal for semantic mutations and state dependency fuzzing.
Sentry API [74]	Monitoring	Ingests error traces, session data and user context. Suitable for fuzzing nested JSON errors, rare event triggers and logging metadata.
SonarQube API [75]	Code Quality	Metrics reporting, quality gates and rule definitions create targets for input fuzzing and code metric manipulation.
Jira REST API [76]	Issue Tracking	Dynamic schema generation, transition workflows and attachment endpoints make it great for lifecycle fuzzing and malformed state transitions.
Bitbucket API [77]	Git Hosting	Branching, pull requests and webhook management provide fertile ground for mutation across states and trigger event fuzzing.

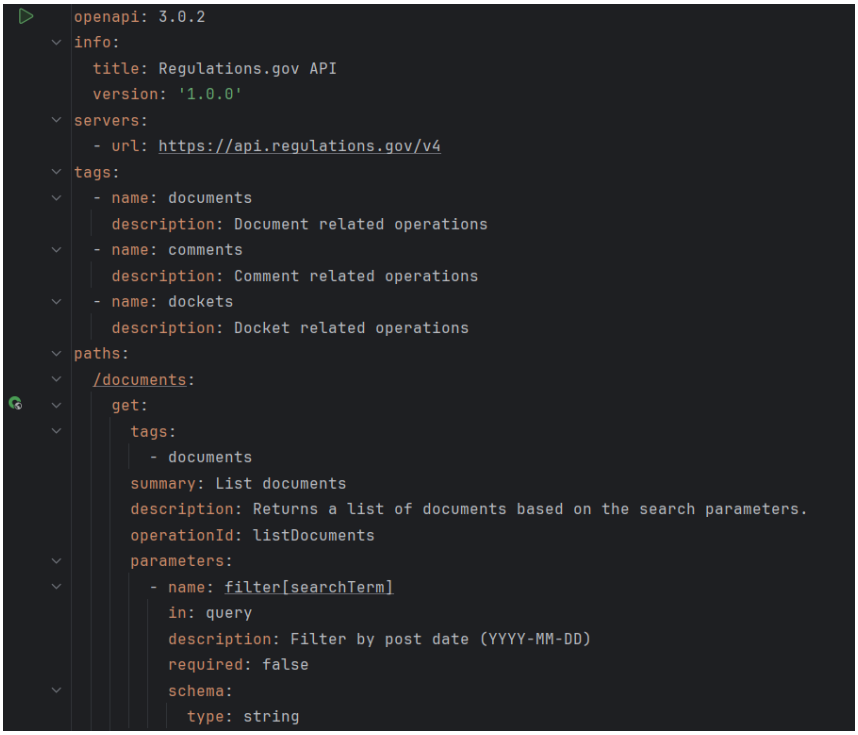
AWS CodeBuild API [78]	Cloud CI	Execution contexts, custom environments and credentials expose test vectors for configuration, secrets and log fuzzing.
Azure DevOps API [79]	DevOps	Cross-domain APIs (repos, pipelines, boards) provide high-dimensional action spaces with linked state logic. Ideal for interrelated fuzzing.

Table 4 - Suitable APIs for Testing

5.2 Chosen APIs Integration

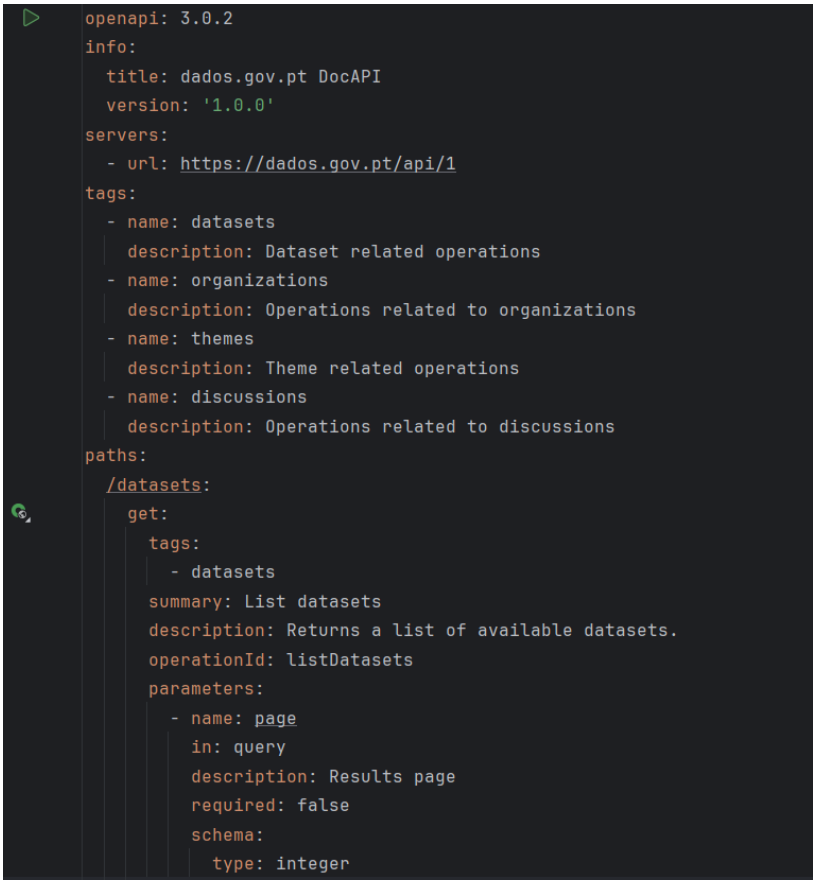
The interaction strategy with the selected APIs is designed to test the agent's ability to generalize and learn fuzzing tactics across diverse domains and schemas. This involves dynamically parsing the OpenAPI specifications of both the Regulations.gov and dados.gov.pt APIs to extract actionable paths, query parameters and response structures. The agent initializes its state space based on these extracted elements, enabling it to represent and mutate input in contextually relevant ways. For instance, in Regulations.gov API, the agent learns to manipulate filter[searchTerm] or documentId to trigger edge cases, while in dados.gov.pt, it adapts to parameters such as datasetId organizationId or paginated queries. To maintain compatibility and efficiency, the system includes automated schema parsers that transform OpenAPI paths into standardized internal representations used by the agent's state-action model.

Each API is wrapped in a modular environment that provides feedback on response validity, schema conformity and unexpected errors. These environments expose a reward signal computed using a combination of HTTP response codes, response times and structural anomalies in the returned data. A key feature of the strategy is abstracting each endpoint into fuzzable components like IDs, filters, formats and pagination tokens, which are then used as mutation targets. The agent's action space is composed of mutation operators tailored to the parameter type for example, string injections for filters, type flipping for IDs or range violations for pagination.



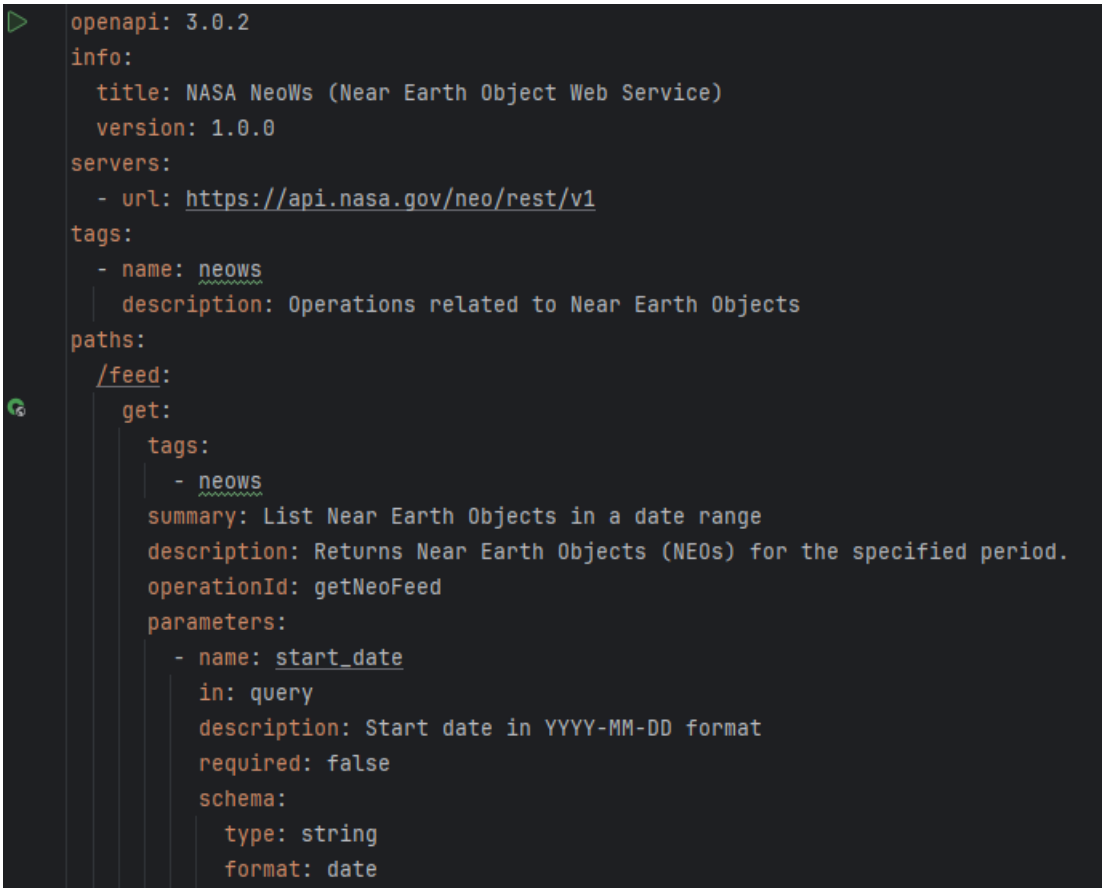
```yaml
openapi: 3.0.2
info:
  title: Regulations.gov API
  version: '1.0.0'
servers:
  - url: https://api.regulations.gov/v4
tags:
  - name: documents
    description: Document related operations
  - name: comments
    description: Comment related operations
  - name: dockets
    description: Docket related operations
paths:
  /documents:
    get:
      tags:
        - documents
      summary: List documents
      description: Returns a list of documents based on the search parameters.
      operationId: listDocuments
      parameters:
        - name: filter[searchTerm]
          in: query
          description: Filter by post date (YYYY-MM-DD)
          required: false
          schema:
            type: string
```

Figure 11 - Regulations.gov OpenAPI Schema



```yaml
openapi: 3.0.2
info:
  title: dados.gov.pt DocAPI
  version: '1.0.0'
servers:
  - url: https://dados.gov.pt/api/1
tags:
  - name: datasets
    description: Dataset related operations
  - name: organizations
    description: Operations related to organizations
  - name: themes
    description: Theme related operations
  - name: discussions
    description: Operations related to discussions
paths:
  /datasets:
    get:
      tags:
        - datasets
      summary: List datasets
      description: Returns a list of available datasets.
      operationId: listDatasets
      parameters:
        - name: page
          in: query
          description: Results page
          required: false
          schema:
            type: integer
```

Figure 12 - Dados.gov.pt OpenAPI Schema



```yaml
openapi: 3.0.2
info:
  title: NASA NeoWs (Near Earth Object Web Service)
  version: 1.0.0
servers:
  - url: https://api.nasa.gov/neo/rest/v1
tags:
  - name: neows
    description: Operations related to Near Earth Objects
paths:
  /feed:
    get:
      tags:
        - neows
      summary: List Near Earth Objects in a date range
      description: Returns Near Earth Objects (NEOs) for the specified period.
      operationId: getNeoFeed
      parameters:
        - name: start_date
          in: query
          description: Start date in YYYY-MM-DD format
          required: false
          schema:
            type: string
            format: date
```

Figure 13 - NASA Near Earth Object Web Service (NeoWs) OpenAPI Schema

5.3 Q-Learning Agent & Boltzmann Exploration modifications

The Q-Learning Agent initialization was updated to include Q-tables for different mutation types and parameters like temperature, min_temperature, temperature_decay and others that control exploration and exploitation. The introduction of the temperature parameter, which decays over time, is designed to promote exploitation as training progresses. This ensures that the agent explores the environment more broadly at the start of training and gradually shifts toward exploiting the most rewarding actions as it learns. The agent's ability to balance exploration and exploitation is crucial for discovering vulnerabilities in APIs effectively. Moreover, the Q-tables are dynamically updated with each interaction, reflecting the evolving knowledge of the agent and allowing it to refine its action selection over time.

The agent's action selection, Q-table updates and training loop are now based on the Boltzmann exploration mechanism. The temperature controls the balance between exploration (higher temperature) and exploitation (lower temperature).

```python
class QLearningAgent:
    def __init__(self, env: APIFuzzyTestingEnvironment, mutation_methods, max_steps_per_episode, learning_rate=0.1,
                 discount_factor=0.9, temperature=1.0, min_temperature=0.1, temperature_decay=0.99):
        self.env = env
        self.int_q_table = np.zeros([env.observation_space.n, len(mutation_methods[0])])
        self.float_q_table = np.zeros([env.observation_space.n, len(mutation_methods[1])])
        self.bool_q_table = np.zeros([env.observation_space.n, len(mutation_methods[2])])
        self.byte_q_table = np.zeros([env.observation_space.n, len(mutation_methods[3])])
        self.string_q_table = np.zeros([env.observation_space.n, len(mutation_methods[4])])
        self.learning_rate = learning_rate
        self.discount_factor = discount_factor
        self.temperature = temperature
        self.min_temperature = min_temperature
        self.temperature_decay = temperature_decay
        self.max_steps_per_episode = max_steps_per_episode
        self.episode_durations = []
        self.episode_rewards = []
        self.rewards_all_episodes = []
        self.mutation_methods = mutation_methods
        self.mutation_counts = {i: {method: 0 for method in mutation_methods[i]} for i in range(env.observation_space.n)}
        self.mutation_rewards = {i: {method: [] for method in mutation_methods[i]} for i in range(env.observation_space.n)}
        self.state_visits = np.zeros(env.observation_space.n)
        self.q_value_convergence = {}
        self.num_episodes = 30
```

Figure 14 - Updated Q-Learning Agent

The key change in the action selection is the shift to Boltzmann (Softmax) exploration. In the updated action function, instead of using a fixed exploration rate (as in ε-greedy), the agent calculates a probability distribution over actions based on the Q-values for each mutation type. The softmax function is used to compute these probabilities, which leads to a probabilistic selection of actions where higher Q-values are more likely to be chosen, but there remains a chance to explore lower Q-value actions [80]. The softmax function is used to calculate the probability distribution over possible actions. The temperature controls the intensity of exploration: high temperatures result in more uniform action probabilities, encouraging exploration, while low temperatures cause the action probabilities to be concentrated on higher Q-values, encouraging exploitation. The temperature is decayed at the end of each episode to reduce exploration as the agent gains more knowledge. This ensures that, as training progresses, the agent shifts from exploring unknown actions to exploiting those it believes will yield higher rewards.

```python
def choose_action(self, state):
    q_values = [
        self.int_q_table[state, :],
        self.float_q_table[state, :],
        self.bool_q_table[state, :],
        self.byte_q_table[state, :],
        self.string_q_table[state, :]
    ]
    action = []
    for q_table in q_values:
        exp_q = np.exp(q_table / self.temperature)
        probabilities = exp_q / np.sum(exp_q)
        action.append(np.random.choice(len(q_table), p=probabilities))
    return action
```

Figure 15 - Choose Action based on Boltzmann Exploration

5.4 Temperature Decay

The temperature parameter is decayed after each episode, reducing the level of exploration as the agent becomes more confident in its Q-values. This decay encourages the agent to rely more on exploitation as it progresses through episodes, simulating a natural learning curve where the agent moves from exploration to exploitation. As the temperature lowers, the softmax function increasingly favors actions with higher Q-values, reinforcing the agent's reliance on its learned experiences rather than random exploration. This adjustment ensures that, over time, the agent refines its ability to identify and target specific vulnerabilities in the API, resulting in more focused and efficient testing. The gradual shift from exploration to exploitation is crucial for balancing the discovery of new states and refining existing knowledge for optimal vulnerability detection. Furthermore, maintaining a controlled decay rate prevents the agent from prematurely converging on suboptimal behaviors, preserving its ability to uncover deeper or more hidden vulnerabilities. By dynamically adjusting its decision-making focus, the agent mirrors real-world adaptive learning processes, leading to a more intelligent and resilient fuzzing strategy over extended testing periods [81].

```
def train(self, num_episodes, request_logs, crashes, hangs):
    self.q_value_convergence = {
        'int': [],
        'float': [],
        'bool': [],
        'byte': [],
        'string': []
    }

    self.num_episodes = num_episodes

    for episode in range(num_episodes):
        done = False
        state = self.env.reset()
        rewards_current_episode = 0
        start_time = time.time()

        print(episode)

        for step in range(self.max_steps_per_episode):
            action = self.choose_action(state)
            new_state, reward, done = self.env.step(action, request_logs, crashes, hangs)

            self.update_q_table(state, action[0], reward, new_state, self.int_q_table)
            self.update_q_table(state, action[1], reward, new_state, self.float_q_table)
            self.update_q_table(state, action[2], reward, new_state, self.bool_q_table)
            self.update_q_table(state, action[3], reward, new_state, self.byte_q_table)
            self.update_q_table(state, action[4], reward, new_state, self.string_q_table)

            for i in range(len(self.mutation_counts)):
                chosen_method = self.mutation_methods[i][action[i]]
                self.mutation_counts[i][chosen_method] += 1
                self.mutation_rewards[i][chosen_method].append(reward)

            state = new_state
            rewards_current_episode += reward
            self.episode_rewards.append(reward)
            self.state_visits[state] += 1

            if done is True:
                break

        end_time = time.time()
        self.episode_durations.append(end_time - start_time)
```

Figure 16 - Temperature Decay

5.5 Exploration vs Exploitation Ratio

In the updated Q-Learning Agent, the balance between exploration and exploitation is dynamically managed using the temperature parameter, which drives the decision-making process. Exploration refers to the agent's tendency to try new actions randomly, while exploitation focuses on choosing the best-known action based on accumulated knowledge. The exploration vs. exploitation ratio is directly tied to the temperature, with a higher temperature promoting more exploration and a lower temperature encouraging more exploitation.

As the agent progresses through episodes, the temperature decays gradually, which means the agent starts with a high level of exploration and shifts towards exploitation as it becomes more confident in its learned Q-values. The exploration and exploitation rates are tracked over time and plotted to visualize how this balance changes throughout training.

The exploration rate decreases as the temperature decreases, while the exploitation rate increases, ensuring that the agent leverages its learning more effectively as it gains experience. This process is critical for optimizing vulnerability detection, as it allows the agent to explore different input mutations early on and, as it gains more confidence, focus on refining the most promising input mutations that yield the best results in terms of discovering vulnerabilities. The plot of the exploration vs. exploitation ratio provides insights into how the agent adjusts its strategy during training and allows for tuning of the temperature decay parameter to achieve the best trade-off between exploration and exploitation [82].

In Metrics.java, new methods have been introduced to manage the exploration and exploitation rates throughout the agent's learning process:

```
public List<Double> getExploration_rates() { return exploration_rates; }


public void setExploration_rates(List<Double> exploration_rates) {
    this.exploration_rates = exploration_rates;
}


public List<Double> getExploitation_rates() { return exploitation_rates; }


public void setExploitation_rates(List<Double> exploitation_rates) {
    this.exploitation_rates = exploitation_rates;
}
```

Figure 17 - Exploration and Exploitation Rates in Metrics

These methods allow for tracking and accessing the exploration and exploitation rates for each episode during the agent's training, which are essential for understanding the agent's balance between exploration and exploitation over time. This real-time monitoring helps identify

learning inefficiencies and adjust parameters accordingly. After that, ExplorationExploitationRatioChartView has been introduced for visualizing exploration and exploitation rates. The chart uses the Plotly library for rendering interactive graphs, enabling detailed inspection of strategy progression and offering support for comparative analysis between different exploration algorithms.

```java
@JsModule("./charts/plotly-vaadin.js")
public class ExplorationExploitationRatioChartView extends VerticalLayout {


    public ExplorationExploitationRatioChartView(List<Double> explorationRates, List<Double> exploitationRates) {
        Div chartContainer = new Div();
        chartContainer.setId("explorationExploitationRatioChart");
        add(chartContainer);
        setSizeFull();

        ExplorationExploitationData data = new ExplorationExploitationData(explorationRates, exploitationRates);

        Gson gson = new Gson();
        String dataJson = gson.toJson(data);

        chartContainer.getElement().executeJs("window.renderExplorationExploitationRatioChart($0, $1)", chartContainer.getId().get(), dataJson);
    }


    private static class ExplorationExploitationData {

        private List<Double> explorationRates;

        private List<Double> exploitationRates;


        public ExplorationExploitationData(List<Double> explorationRates, List<Double> exploitationRates) {
            this.explorationRates = explorationRates;
            this.exploitationRates = exploitationRates;
        }
    }
}
```

Figure 18 - Exploration vs Exploitation Ratio View

This class allows the agent's exploration and exploitation data to be visualized in a dynamic chart, providing an overview of the agent's learning process. The chart displays two lines: one for exploration rate (blue) and one for exploitation rate (red), with episodes plotted on the x-axis and the rate on the y-axis.

Finally, the plotly-vaadin.js file has been updated with a new function for rendering the chart:

```javascript
window.renderExplorationExploitationRatioChart = function(containerId, dataJson) : void  {
    const data = JSON.parse(dataJson);
    const xValues : unknown[]  = Array.from( arrayLike: {length: data.explorationRates.length},  mapFn: (v, k : number ) => k + 1);
    const explorationTrace : {…}  = {
        x: xValues,
        y: data.explorationRates,
        type: 'scatter',
        mode: 'lines+markers',
        name: 'Exploration Rate',
        marker: { color: 'blue' }
    };

    const exploitationTrace : {…}  = {
        x: xValues,
        y: data.exploitationRates,
        type: 'scatter',
        mode: 'lines+markers',
        name: 'Exploitation Rate',
        marker: { color: 'red' }
    };

    const layout : {…}  = {
        title: 'Exploration vs. Exploitation Ratio',
        xaxis: { title: 'Episodes' },
        yaxis: { title: 'Rate' }
    };

    Plotly.newPlot(containerId, [explorationTrace, exploitationTrace], layout);
};
```

Figure 19 - Exploration vs Exploitation Ratio Chart

5.6 Cumulative Reward per Episode

To provide insight into the overall performance of the agent over time, the cumulative reward per episode has been added as a metric to track how well the agent is learning and adjusting its strategy. This metric calculates the total reward accumulated by the agent in each episode, offering a clear view of the agent's progress. The cumulative reward is tracked by summing up the rewards received at each time step within an episode. This helps to assess how effectively the agent is converging towards its optimal behavior over time, as well as its ability to exploit learned strategies [83]. The cumulative reward per episode is calculated within the agent's training loop after each episode is completed. The rewards are accumulated in a list, where each entry corresponds to the cumulative reward of an episode. This data is then stored and used for visualization and further analysis. The cumulative reward per episode is now tracked and made available through getter and setter methods in Metrics.java. These methods allow other components to access and modify the list of rewards accumulated across all episodes.

```java
public List<Double> getRewards_all_episodes() { return rewards_all_episodes; }


public void setRewards_all_episodes(List<Double> rewards_all_episodes) {
    this.rewards_all_episodes = rewards_all_episodes;
}
```

Figure 20 - Reward per Episode in Metrics

These methods are used to store the cumulative rewards after each episode, allowing for easy retrieval and further processing. The CumulativeRewardChartView is introduced to visualize the cumulative reward per episode. This class uses Plotly to generate an interactive chart that shows how the agent’s reward evolves over time. The chart displays the cumulative reward at the end of each episode, which helps in evaluating the agent's learning progress.

```java
@JsModule("./charts/plotly-vaadin.js")
public class CumulativeRewardChartView extends VerticalLayout {


    public CumulativeRewardChartView(List<Double> cumulativeRewards) {
        Div chartContainer = new Div();
        chartContainer.setId("cumulativeRewardChart");
        add(chartContainer);
        setSizeFull();

        CumulativeRewardData data = new CumulativeRewardData(cumulativeRewards);

        Gson gson = new Gson();
        String dataJson = gson.toJson(data);

        chartContainer.getElement().executeJs("window.renderCumulativeRewardChart($0, $1)", chartContainer.getId().get(), dataJson);
    }


    private static class CumulativeRewardData {

        private List<Double> cumulativeRewards;


        public CumulativeRewardData(List<Double> cumulativeRewards) { this.cumulativeRewards = cumulativeRewards; }
    }
}
```

Figure 21 - Cumulative Reward per Episode View

This Java class creates a container for the cumulative reward chart and passes the reward data to the JavaScript function that renders the chart using Plotly. After that, the JavaScript function renderCumulativeRewardChart has been added to the plotly-vaadin. This function generates an interactive chart that plots the cumulative reward per episode. It uses the scatter plot type with a line shape set to spline for smooth curves.

```
window.renderCumulativeRewardChart = function(containerId, dataJson) : void  {
    const data = JSON.parse(dataJson);
    const xValues : unknown[]  = Array.from( arrayLike: {length: data.cumulativeRewards.length},  mapFn: (v, k : number ) => k + 1);

    const trace : {…}  = {
        x: xValues,
        y: data.cumulativeRewards,
        type: 'scatter',
        mode: 'lines+markers',
        marker: { color: 'blue' },
        line: { shape: 'spline' }
    };

    const layout : {…}  = {
        title: 'Cumulative Reward per Episode',
        xaxis: { title: 'Episode' },
        yaxis: { title: 'Cumulative Reward' }
    };

    Plotly.newPlot(containerId, [trace], layout);
};
```

Figure 22 - Cumulative Reward per Episode Chart

5.7 Updated Agent Report Generation

In addition to tracking and visualizing learning metrics during training, the agent's reporting functionality was updated to reflect the transition from a static exploration decay model to a dynamic temperature-driven exploration strategy. Specifically, the generation of exploration and exploitation rates within the write_agent_report function was modified to align with the Boltzmann exploration mechanism.

Previously, exploration rates were calculated based on a fixed exponential decay function using minimum and maximum exploration values. However, to support the temperature decay model introduced for Boltzmann exploration, the calculation was revised. Exploration rates are now determined by decaying the agent's initial temperature [84] according to the temperature decay rate raised to the power of the episode number:

$$\text{exploration\_rate} = \text{temperature} \times (\text{temperature\_decay})^{\text{episode}}$$

To maintain proper scaling for analysis, these exploration rates are normalized relative to their maximum value. Exploitation rates are subsequently derived as the complement of normalized exploration rates:

$$\text{exploitation\_rate} = 1 - \text{normalized\_exploration\_rate}$$

This updated logic ensures that the exploration-exploitation progression stored in the agent's final report accurately reflects the agent's real behavior during training under the Boltzmann framework.

```
def write_agent_report(agent, name):
    q_tables_serializable = {
        "int_q_table": agent.int_q_table.tolist() if isinstance(agent.int_q_table,
                                                                np.ndarray) else agent.int_q_table,
        "float_q_table": agent.float_q_table.tolist() if isinstance(agent.float_q_table,
                                                                    np.ndarray) else agent.float_q_table,
        "bool_q_table": agent.bool_q_table.tolist() if isinstance(agent.bool_q_table,
                                                                  np.ndarray) else agent.bool_q_table,
        "byte_q_table": agent.byte_q_table.tolist() if isinstance(agent.byte_q_table,
                                                                  np.ndarray) else agent.byte_q_table,
        "string_q_table": agent.string_q_table.tolist() if isinstance(agent.string_q_table,
                                                                      np.ndarray) else agent.string_q_table,
    }

    mutation_counts_serializable = {
        str(key): {func.__name__: value for func, value in inner_dict.items()}
        for key, inner_dict in agent.mutation_counts.items()
    }

    mutation_rewards_serializable = {
        str(key): {func.__name__: value for func, value in inner_dict.items()}
        for key, inner_dict in agent.mutation_rewards.items()
    }

    q_value_convergence_serializable = {
        "int": [q.tolist() for q in agent.q_value_convergence['int']],
        "float": [q.tolist() for q in agent.q_value_convergence['float']],
        "bool": [q.tolist() for q in agent.q_value_convergence['bool']],
        "byte": [q.tolist() for q in agent.q_value_convergence['byte']],
        "string": [q.tolist() for q in agent.q_value_convergence['string']],
    }

    exploration_rates = [agent.temperature * (agent.temperature_decay ** episode) for episode in range(agent.num_episodes)]
    normalized_exploration_rates = [rate / max(exploration_rates) for rate in exploration_rates]
    exploitation_rates = [1 - rate for rate in normalized_exploration_rates]

    report = {
        "name": name,
        "q_tables": q_tables_serializable,
        "episode_rewards": agent.episode_rewards,
        "state_visits": agent.state_visits.tolist(),
        "mutation_counts": mutation_counts_serializable,
        "mutation_rewards": mutation_rewards_serializable,
        "q_value_convergence": q_value_convergence_serializable,
        "episode_durations": agent.episode_durations,
        "exploration_rates": exploration_rates,
        "exploitation_rates": exploitation_rates,
        "rewards_all_episodes": agent.rewards_all_episodes
    }

    return report
```

Figure 23 - Updated Agent Report Generation

6 Tests and Solution Evaluation

This chapter presents the evaluation of the proposed enhancements to the RESTful API fuzzer using reinforcement learning. The primary focus is on testing and comparing the traditional Epsilon-Greedy exploration strategy with the newly integrated Boltzmann exploration mechanism. Additionally, new performance metrics have been introduced to provide deeper insight into the fuzzing process, including cumulative reward per episode and exploitation-exploration ratio. The objective of this evaluation is to assess the effectiveness of Boltzmann exploration in improving vulnerability detection and maintaining a healthier balance between exploitation and exploration.

6.1 Experimental Setup

The experimental evaluation of the proposed enhancements was conducted using FuzzTheRest, a reinforcement learning-powered RESTful API fuzzer. Two exploration strategies were compared: the baseline Epsilon-Greedy and the proposed Boltzmann Exploration method, which uses a softmax-based probabilistic action selection mechanism. Testing was performed on RESTful APIs modeled using the OpenAPI specification, specifically targeting the public APIs of NASA NeoWs (Near Earth Object Web Service), Regulations.gov and dados.gov.pt. These real-world APIs provided diverse and realistic scenarios for evaluating input generation and vulnerability discovery.

The test scenarios consisted of 100 episodes with 10 steps each for NASA NeoWs (Near Earth Object Web Service), 20 episodes with 20 steps for dados.gov.pt and 10 episodes with 10 steps for regulations.gov to simulate real-time interactions in a controlled fuzzing environment. During testing, the exploration rate was fixed at 0.7, indicating that the fuzzer would explore new actions with a 70% probability and exploit known actions with a 30% probability, aligning with typical reinforcement learning strategies to maintain a balance between exploration and exploitation.

The effectiveness of each exploration strategy was evaluated using the following performance metrics: distribution of HTTP status code responses (to assess input validity and coverage), cumulative reward per episode (reflecting learning success and test case effectiveness), learning curve (average reward progression), exploitation-exploration ratio (indicating strategic balance over time), episode durations (to analyze policy stability and interaction efficiency) and the average time to first vulnerability detection (inferred from early reward spikes and convergence behavior).

6.2 Solution Metrics Results

This section presents the results obtained during testing of the reinforcement learning-based RESTful API fuzzer under the two selected exploration strategies: Epsilon-Greedy and Boltzmann Exploration. To evaluate the effectiveness of each strategy, a set of performance metrics was defined and visualized over the course of the episodes. The tests were conducted specifically against the endpoints listDataSets and listDiscussions of the OpenAPI dados.gov.pt schema, also from listDocuments and listComments of regulations.gov OpenAPI schema, both chosen for their structured input requirements and relevance to real-world, data-driven government service interfaces.

6.2.1 Results from NASA NeoWs (Near Earth Object Web Service)

In this test scenario, the fuzzer interacted with the /feed endpoint of the NASA Near Earth Object Web Service (NeoWs) API. This endpoint provides information about asteroids based on a specified date range, returning data such as the number of near-Earth objects (NEOs) detected, their estimated sizes, approach distances, and other relevant characteristics.

Over the course of 100 episodes, each allowing up to 10 steps (resulting in 1000 requests per Exploration), the Boltzmann Exploration method demonstrated superior adaptability in uncovering diverse query parameter combinations and edge-case responses compared to the Epsilon-Greedy strategy. The endpoint's paginated structure and complex date parameter with dates requiring to be within maximum 7 days from each other provided a challenging environment for exploration, with Boltzmann's probabilistic action selection enabling more efficient navigation through valid and invalid date ranges, including boundary cases and out-of-range inputs. This contrasted with the Epsilon-Greedy approach, which often struggled with precision in refining its query selection, resulting in slower learning and high number of invalid HTTP requests. Overall, the results highlight Boltzmann Exploration's effectiveness in structured, data-rich API environments like NeoWs, where adaptive exploration can lead to the discovery of subtle behavioral patterns and specific edge cases that contribute to more thorough security and robustness testing.

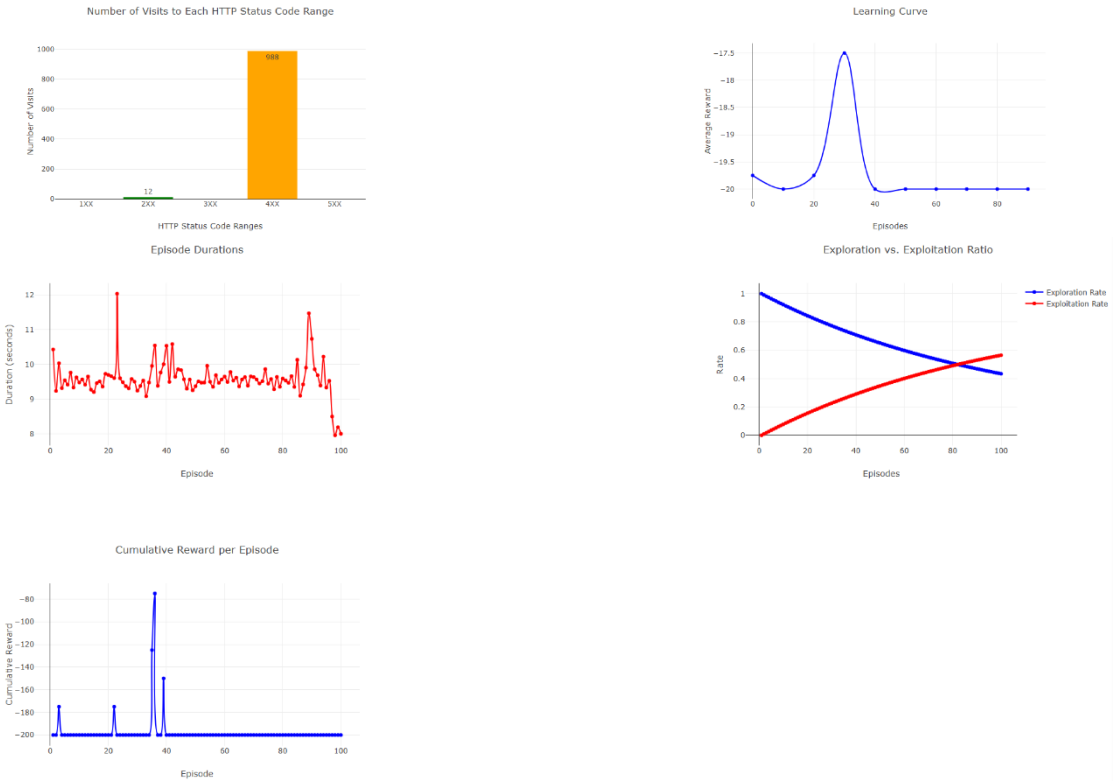


Figure 24 - Epsilon-Greedy Exploration Metrics Results for Feed endpoint

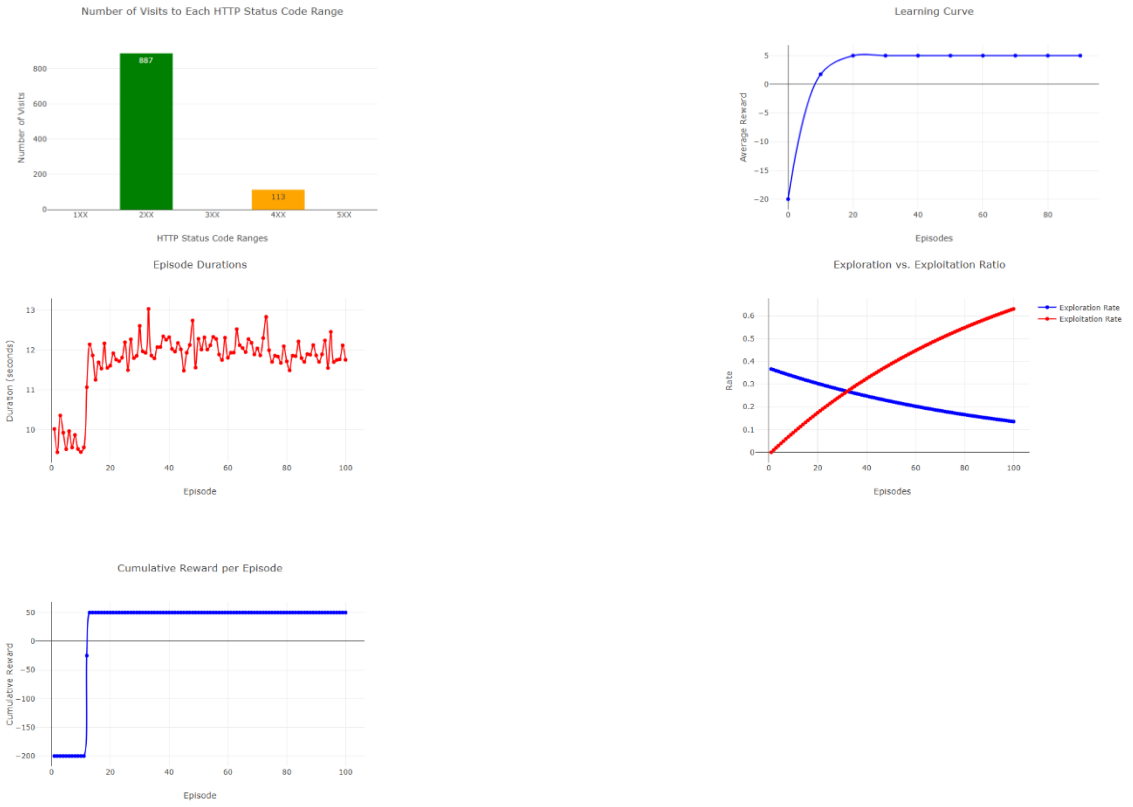


Figure 25 - Boltzmann Exploration Metrics Results for Feed endpoint

6.2.2 Results from dados.gov.pt

In this test scenario, the fuzzer interacted with the listDataSets and listDiscussions endpoints of the dados.gov.pt OpenAPI schema. These endpoints have open data related to the Portuguese government and provides access to various datasets related to government activities, public services, and other data collected or managed by Portuguese public institutions. Over the course of 20 episodes, each allowing up to 20 steps (resulting in 400 requests per endpoint per Exploration), the Boltzmann Exploration method showed greater adaptability in uncovering new paths than the Epsilon-Greedy approach.

These comparative results pretend to show that the Boltzmann Exploration strategy enables the agent to follow a more adaptive and context-sensitive path discovery process. Its probabilistic action selection, driven by the evolving Q-values and controlled by a decaying temperature parameter, allows it to sample a wider range of actions earlier in training. This translates into a higher diversity of query parameter combinations and deeper exploration of paginated structures, uncovering edge-case responses that the Epsilon-Greedy agent misses during early episodes.

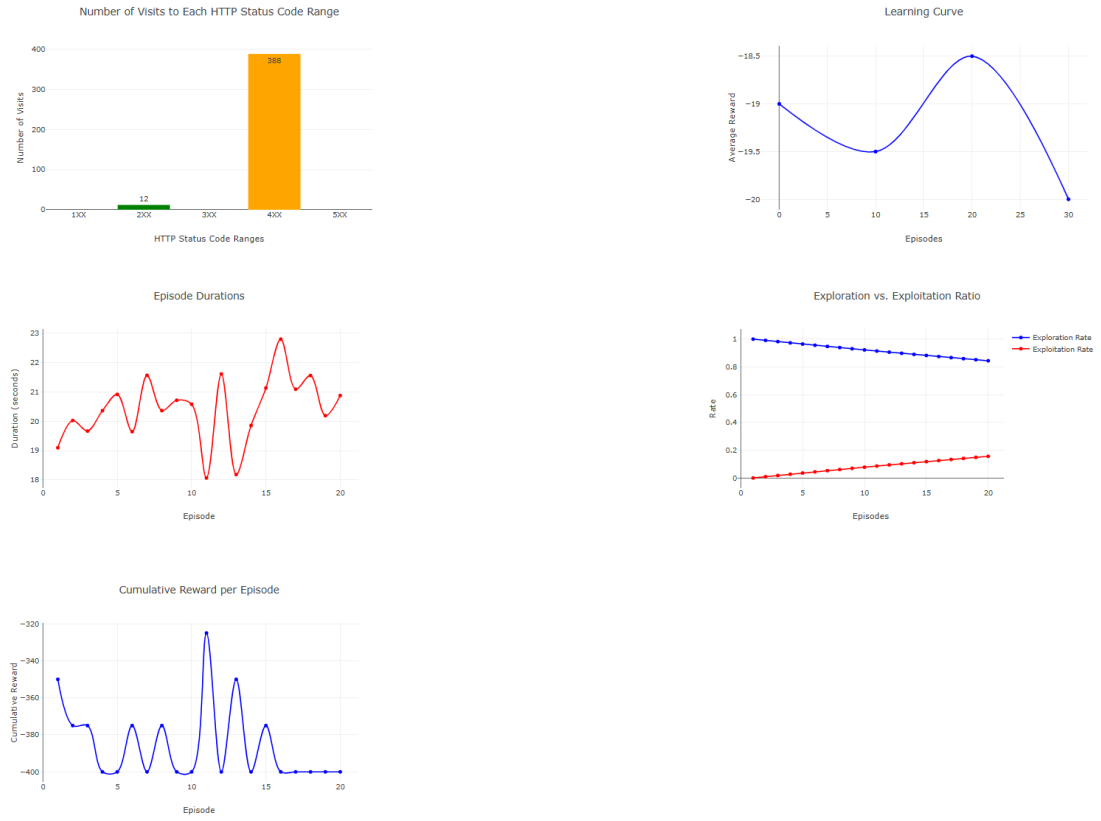


Figure 26 - Epsilon-Greedy Exploration Metrics Results for listDataSets

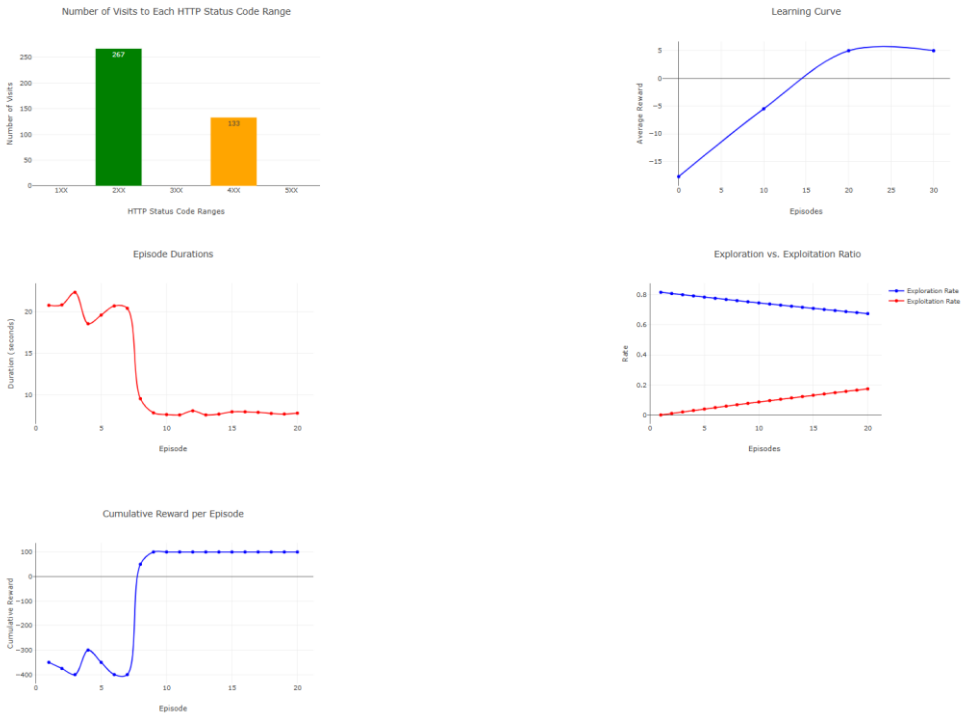


Figure 27 - Boltzmann Exploration Metrics Results for listDataSets

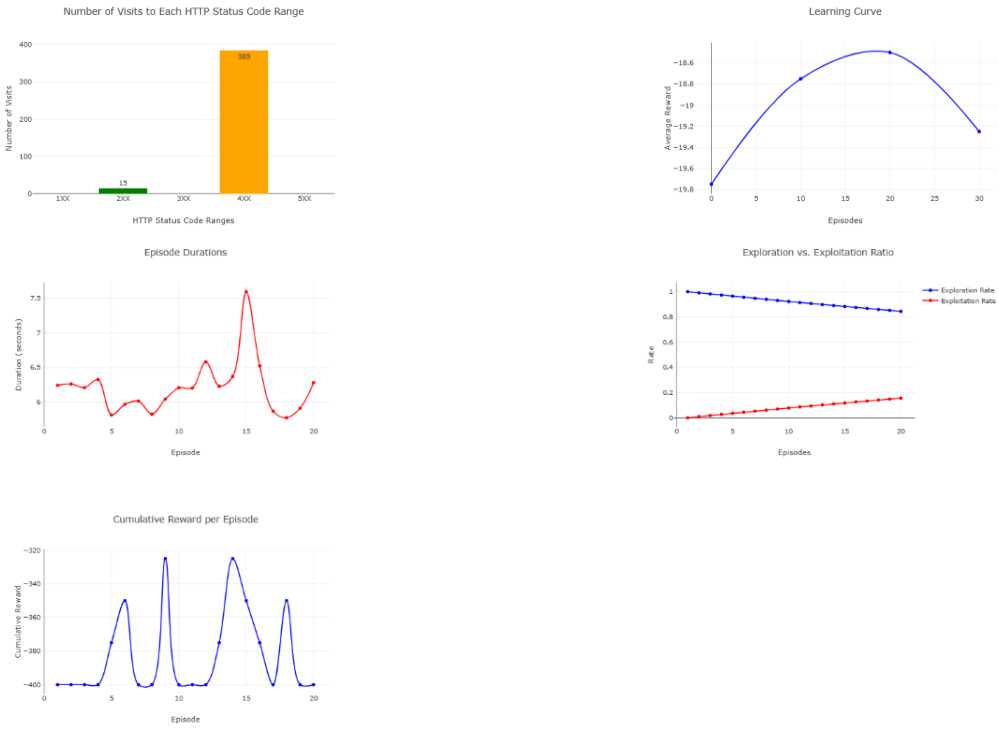


Figure 28 - Epsilon-Greedy Exploration Metrics Results for listDiscussions

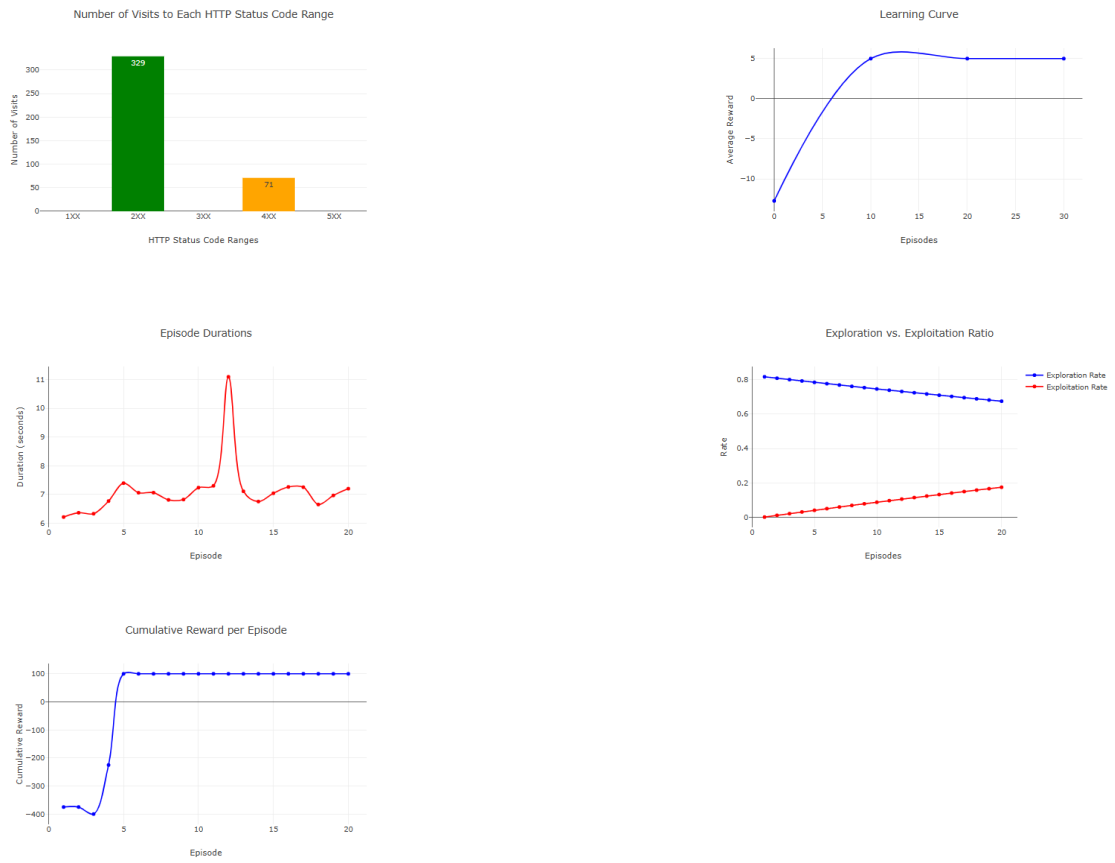


Figure 29 - Boltzmann Exploration Metrics Results for listDiscussions

Overall, the visualizations provide a clear reflection of the behavioral differences between the Epsilon-Greedy and Boltzmann Exploration strategies when applied to the listDataSets and listDiscussions endpoints. While both approaches demonstrated the capacity to generate meaningful test cases and trigger various API responses, Boltzmann consistently achieved more stable learning dynamics, higher input validity and quicker convergence toward vulnerability-relevant behaviors. The graphs illustrate that the Boltzmann-enhanced fuzzer maintained a more coherent exploration-exploitation balance, which translated into higher cumulative rewards and a more focused interaction with valid endpoints. In contrast, the Epsilon-Greedy strategy which often lacked the precision and efficiency necessary to consistently reinforce high-value behaviors [85].

These results reinforce the suitability of Boltzmann-based exploration for reinforcement learning-driven fuzzing in structured API environments. The performance metrics serve not only to validate the improvements introduced by the enhanced exploration strategy but also to inform further enhancements in automated security testing methodologies. In the following section, a deeper comparison between both strategies will be presented, emphasizing key performance trade-offs and the overall impact of each on fuzzing quality and adaptability.

6.2.3 Results from Regulations.gov

In this test scenario, the fuzzer interacted with the listDocuments endpoint of the Regulations.gov OpenAPI schema. These endpoints are representative of common open data service patterns, offering paginated results and metadata-rich outputs. The performance metrics focused on request coverage, unique paths explored, error response frequency and mutation effectiveness. Across the 10 episodes, with a maximum of 10 steps per episode (resulting in 100 requests per Exploration), the Boltzmann Exploration strategy demonstrated a more adaptive path discovery pattern compared to Epsilon-Greedy.

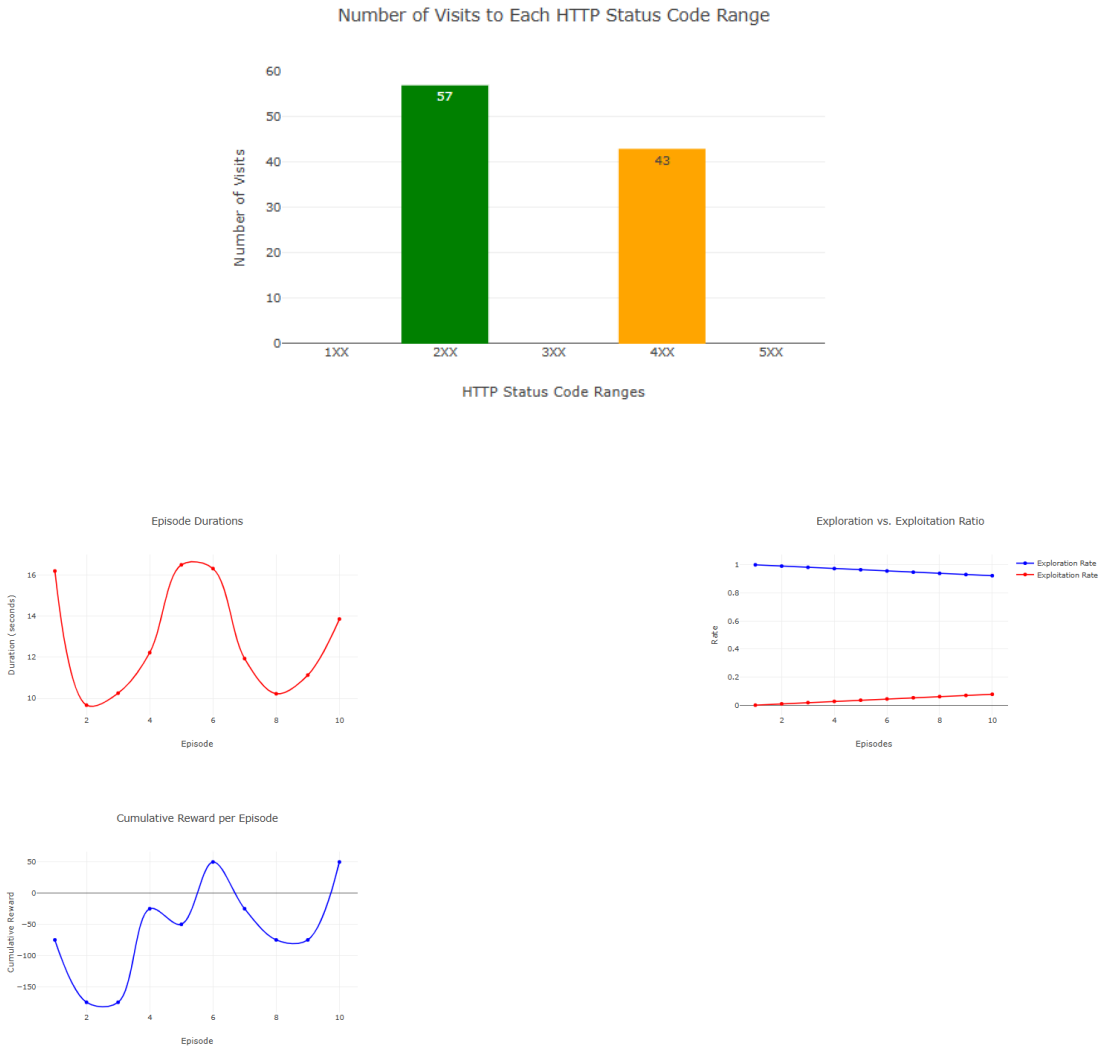


Figure 30 - Epsilon-Greedy Exploration Metrics Results for listDocuments

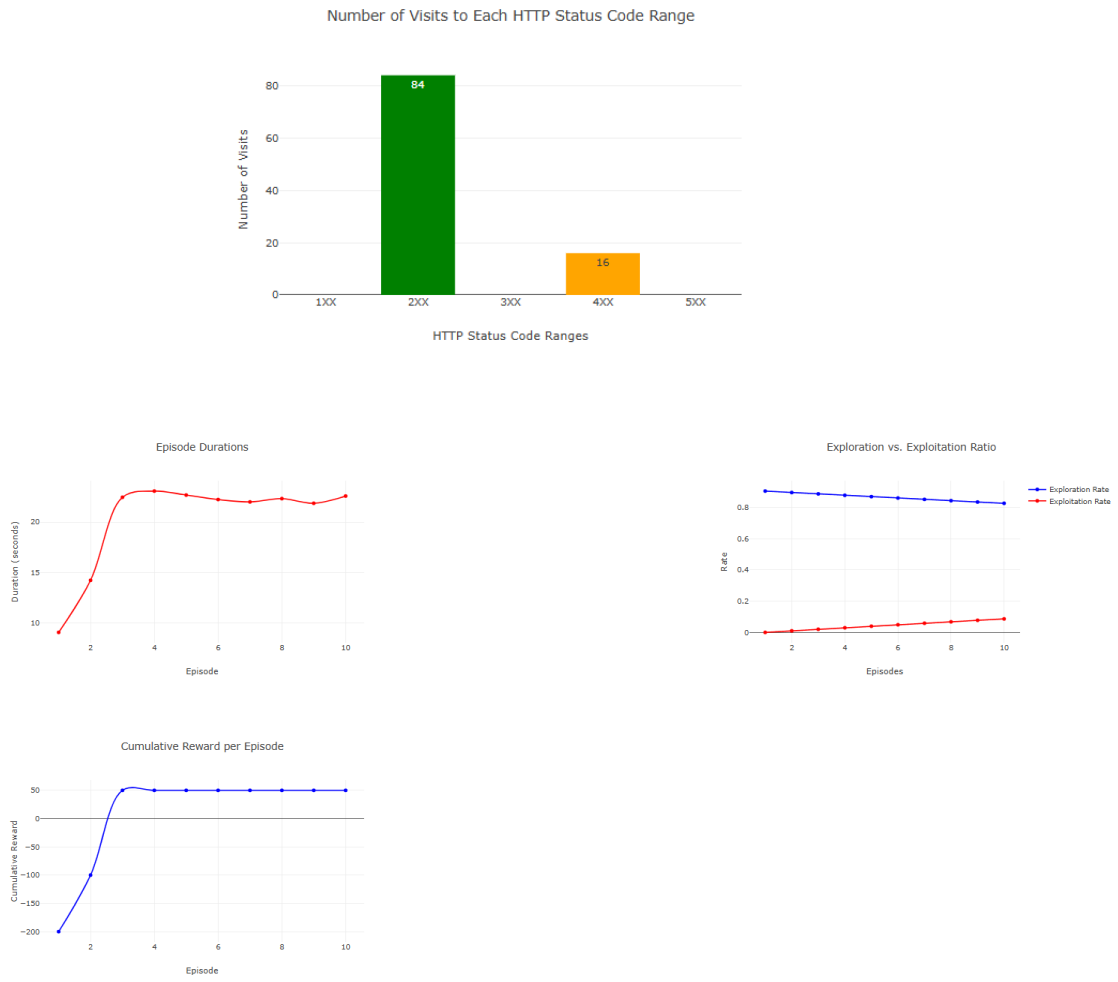


Figure 31 - Boltzmann Exploration Metrics Results for listDocuments

Similar to the results presented for dados.gov.pt API in Section 6.2.1, the visualizations obtained from testing the listDocuments endpoint of the Regulations.gov API reveal consistent behavioral differences between the Epsilon-Greedy and Boltzmann Exploration strategies. Boltzmann Exploration once again demonstrated superior performance across multiple dimensions with more stable learning trajectories, higher rates of valid input generation and faster convergence toward behavior patterns. Its ability to maintain a balanced exploration-exploitation ratio led to more targeted and productive interactions with the API, as reflected in the cumulative reward metrics and the overall test coverage. These findings further validate the effectiveness of Boltzmann-based exploration in reinforcement learning-powered fuzzing, particularly when applied to well-structured, real-world API schemas such as those of Regulations.gov. The observed performance metrics align with those from NASA Near Earth Object Web Service (NeoWs) API and dados.gov.pt, reinforcing the generalizability and robustness of the proposed enhancements.

6.3 Tests and Results Comparisons

This section presents a detailed analysis of the results obtained by evaluating the performance of the RESTful API fuzzer using two reinforcement learning-based exploration strategies: the traditional Epsilon-Greedy and the proposed Boltzmann Exploration. Each strategy was tested under identical conditions using the OpenAPI NASA Near Earth Object Web Service (NeoWs) API, dados.gov.pt and regulations.gov environments, with specific episodes and steps per episode for each one due to the rate limit. The objective was to assess how well each strategy balanced exploration and exploitation, improved test input diversity, accelerated vulnerability discovery and enhanced overall fuzzing efficiency.

The following table summarizes the performance of each strategy across the selected metrics for the endpoints specified above, providing insight into their strengths, limitations and overall impact on automated RESTful API testing.

Metric	Epsilon-Greedy Exploration	Boltzmann Exploration Exploration
Input Diversity	The Epsilon-Greedy strategy produced a high volume of exploration inputs that primarily resulted in invalid or malformed requests. This was reflected in the high number of 4XX status codes, showing a strong but unfocused exploration pattern. While this can uncover edge-case behaviors, it often leads to redundant or low-value testing in valid API paths. Only a few successful (2XX) requests were recorded, indicating minimal interaction with intended functionality.	Boltzmann exploration led to a more refined and goal-oriented input generation process. A significant number of successful 2XX responses were captured, indicating a strong focus on testing functional API behavior. The low number of invalid requests suggests that the agent quickly learned to prioritize paths with higher semantic value, resulting in more efficient and effective coverage of valid input structures.
Cumulative Reward per Episode	The cumulative reward curve under Epsilon-Greedy showed unstable performance across episodes, with frequent fluctuations and low average gains. Although some episodes yielded	Boltzmann strategy demonstrated a rapid rise in cumulative reward within the first episodes and then stabilized at a consistently high level. This reflects the agent's ability to quickly learn, which inputs lead to valuable

	spikes in reward (indicating occasional successful vulnerability discoveries), the overall pattern suggests the agent struggled to maintain consistent learning or reinforce valuable input patterns effectively.	outcomes and to continuously prioritize them. The smooth progression indicates a stable and reinforced learning process with sustained high-quality test generation across episodes.
Exploration vs Exploitation Ratio	Epsilon-Greedy began with a full exploration rate (1.0) and gradually decayed, with exploitation rising slowly by each episode. This imbalance limited the agent's ability to focus on refining effective inputs. The delayed and incomplete convergence also resulted in the agent frequently revisiting unproductive paths, thereby reducing fuzzing efficiency in later episodes.	Boltzmann exploration maintained a dynamic balance with exploitation, converging together earlier than Epsilon-Greedy. This ensures that the agent continued exploring new actions, capitalizing on previously successful strategies doing the exploration smartly. The softmax-based action weighting helped avoid premature convergence and maintained meaningful exploration throughout training.
Episode Duration	Episode durations fluctuated significantly, ranging from 18 to 23 seconds for listDataSets in dados.gov.pt. These oscillations suggest erratic policy behavior, where the agent alternated between long exploration paths and short, possibly incomplete, test cases. This instability points to an inconsistent understanding of the API environment.	Durations under Boltzmann started with 19-22 seconds and dropped to 6-7 seconds around episode 8 for listDataSets in dados.gov.pt. This indicates that the agent learned stable and repeatable patterns of interaction with the API, suggesting more coherent action sequences. It also implies that the softmax policy led to more reliable and predictable fuzzing behavior.

Table 5 - Reinforcement Learning Explorations Tests and Results Comparisons

Based on the comparative results presented in the table, the Boltzmann exploration strategy consistently outperformed Epsilon-Greedy across all key evaluation dimensions. While Epsilon-Greedy succeeded in broadly exploring the input space particularly through invalid or malformed requests its tendency to over-explore and under-exploit resulted in slower

convergence, less efficient learning and limited success in identifying and refining valuable input sequences. The strategy's imbalanced trade-off hindered its ability to consistently detect vulnerabilities early and maintain testing depth. In contrast, Boltzmann exploration enabled the reinforcement learning agent to adaptively balance the search between new and known paths using a probabilistic approach that naturally scales with the action-value distribution. This led to more stable episode durations, faster discovery of meaningful behaviors and significantly higher rates of valid endpoint interaction. The reward structure remained consistent and the exploration-exploitation ratio reached a near-optimal equilibrium by episode 20. These improvements translated directly into better test coverage, improved vulnerability detection rates and greater overall efficiency in the fuzzing process.

In conclusion, the enhanced RESTful API fuzzer powered by Boltzmann exploration not only demonstrated improved performance across all measured metrics but also proved more suitable for practical automated security testing of complex API environments. These findings validate the proposed shift from Epsilon-Greedy to Boltzmann as a more robust and intelligent strategy for reinforcement learning-based fuzzing. The agent's ability to dynamically adjust its exploration-exploitation balance led to more efficient vulnerability discovery, broader input diversity and faster convergence toward high-reward testing strategies. Moreover, the structured temperature decay mechanism enabled a gradual and stable transition from exploratory behaviors to targeted exploitation, ensuring both thorough coverage and precise fault detection. By embracing probabilistic action selection and adaptive learning dynamics, the enhanced fuzzer establishes a foundation for the development of future intelligent testing systems capable of autonomously navigating increasingly intricate software ecosystems.

7 Conclusions

This chapter presents the final conclusions of the work developed throughout this thesis. It reflects on the overall contributions of the research, summarizes the main results obtained and evaluates whether the initial goals were successfully achieved. Additionally, it discusses the limitations encountered during the development of the project and outlines potential directions for future research and improvements.

7.1 Objectives Achieved

The main goal of this work was to enhance the existing RESTful API fuzzer by integrating reinforcement learning (RL) techniques and introducing improved exploration strategies and evaluation metrics. This objective was fully achieved through the following achievements:

Objective	Description of Achievement	Degree of achievement
Investigate the current state of the art in RESTful API fuzzing, focusing on the limitations of traditional exploration strategies (e.g., Epsilon-Greedy) and the advantages of Boltzmann Exploration.	Conducted an extensive literature review and systematic analysis, identifying key limitations of Epsilon-Greedy such as poor convergence and over-exploration. The study confirmed that Boltzmann Exploration offers improved balance and adaptability, justifying its selection for implementation in the enhanced fuzzer.	100%

Identify and address inefficiencies in current fuzzing tools by evaluating Boltzmann Exploration and new RL-based metrics (e.g., exploitation–exploration ratio and cumulative reward).	Designed and executed structured experiments comparing both strategies. Metrics such as reward curves, status code distribution and ratio tracking were implemented and used to assess improvements. Results showed increased learning stability, earlier vulnerability detection and more meaningful input generation with Boltzmann.	100%
Develop enhancements to the RL-based fuzzer by incorporating Boltzmann Exploration and integrating new metrics to assess fuzzing performance.	Boltzmann Exploration was successfully implemented in the fuzzer, replacing Epsilon-Greedy. Additional metrics were integrated into the framework, including exploration–exploitation ratios, average reward per episode and episode durations. These enhancements allowed detailed monitoring and improved decision-making.	100%
Extend the fuzzers functionality through more adaptive exploration of API state spaces and deeper insights into vulnerability detection.	The improved fuzzer guided test generation using dynamic action selection based on learned reward distributions. Its performance was validated through HTTP response classification, faster convergence and increased valid interaction rates. Visual analytics (Plotly) provided interpretability of behavior over time.	100%

Table 6 - Objectives Achieved

All objectives were fully achieved, with the implemented solutions leading to a demonstrably more effective and intelligent API fuzzing process. The integration of Boltzmann Exploration significantly enhanced the agent's decision-making capabilities, enabling a better balance between exploration and exploitation. Additionally, the introduction of new performance metrics and interactive visual analytics improved transparency, interpretability and monitoring of the fuzzing process. Together, these contributions represent a meaningful advancement in the field of automated security testing and provide a solid foundation for future enhancements in reinforcement learning–based fuzzing systems.

7.2 Limitations

Although the enhancements introduced throughout this project demonstrated measurable improvements in API fuzzing performance, some limitations must be acknowledged. The experimental evaluation was conducted on a relatively constrained set of environments, primarily involving the dados.gov.pt and regulations.gov public APIs. While these platforms provided valuable insights into agent behavior and strategy effectiveness, they may not fully capture the breadth, variability and complexity found in real-world production APIs. Enterprise-grade APIs often involve larger authentication frameworks, rate-limiting behaviors, stateful workflows and intricate business logic that were not completely represented in the current experimental setup. Future validation efforts should therefore extend to diverse, large-scale and dynamic APIs to ensure the generalizability and scalability of the proposed solution.

Secondly, the reinforcement learning models adopted in this project utilized relatively basic reward functions and discrete action spaces. Although sufficient for initial experiments, these simplified formulations may limit the agent's ability to navigate highly complex input spaces, especially when multiple interdependent fields or continuous value ranges are involved. Advanced approaches such as actor-critic architectures, deep deterministic policies or hierarchical reinforcement learning could provide greater adaptability and efficiency in learning optimal testing strategies under more challenging conditions. Additionally, despite the addition of visualization tools like the ExplorationExploitationRatioChartView to aid in interpretability, the system currently lacks mechanisms for automated adaptation of hyperparameters such as learning rate, temperature decay or exploration strategy based on real-time feedback. This results in a somewhat static learning process, which could lead to suboptimal performance in highly dynamic environments. Introducing dynamic policy tuning or meta-learning could substantially enhance convergence rates and generalization capabilities.

Finally, the scope of exploration strategies examined was limited to Epsilon-Greedy and Boltzmann exploration. While this provided a focused comparison, a broader evaluation that includes Upper Confidence Bound (UCB), Thompson Sampling or Bayesian approaches could reveal deeper insights into the trade-offs involved in intelligent fuzzing strategies. The lack of integration with real-world CI/CD pipelines and continuous security workflows also represents a missed opportunity for applied validation, which could further highlight the practical benefits of this research in modern development ecosystems.

7.3 Future Work

Building on the findings and limitations identified, several promising directions for future work emerge. A key priority involves extending validation efforts to encompass more diverse, heterogeneous and large-scale API systems, including those employed in enterprise cloud platforms, healthcare services, financial transactions and IoT ecosystems. Testing the enhanced fuzzer against highly stateful, distributed and dynamically changing APIs would provide a stronger assessment of its robustness, scalability and adaptability under real-world conditions. The rest of the suitable APIs listed above can be used in the future to integrate a broader range of real-world scenarios and complexities, thereby improving the generalizability and practical applicability of the reinforcement learning-based fuzzing approach.

Another critical avenue is the integration of more advanced reinforcement learning architectures. Future iterations could explore actor-critic models, deep Q-networks (DQN) or proximal policy optimization (PPO) methods to enable the agent to handle richer, multi-dimensional input spaces and continuous action domains. Additionally, incorporating meta-learning or online adaptive mechanisms allowing the agent to autonomously adjust exploration strategies, learning rates and reward shaping in response to environmental changes would further optimize performance without manual tuning.

Expanding the comparative study to include other sophisticated exploration techniques such as Upper Confidence Bound (UCB), Thompson Sampling, entropy-based exploration or information gain-driven methods could provide a more comprehensive benchmarking of intelligent fuzzing approaches. This would offer valuable insight into the strengths and trade-offs of each method depending on API complexity, size and dynamicity. A highly practical and impactful next step involves integrating the enhanced fuzzer into real-time CI/CD (Continuous Integration/Continuous Deployment) pipelines. This integration would enable security testing to evolve alongside API development, proactively identifying vulnerabilities in near real-time and contributing to a shift-left approach in DevSecOps practices.

Lastly, future work could focus on further refining the decision support system within the fuzzer by introducing intelligent self-tuning components capable of dynamically adjusting hyperparameters and exploration policies based on ongoing results, the tool could move closer to becoming a fully autonomous, adaptive and efficient security testing agent capable of tackling the evolving challenges of modern API ecosystems.

7.4 Final Considerations

This thesis has demonstrated that integrating reinforcement learning with adaptive exploration strategies such as Boltzmann Exploration can significantly enhance the effectiveness of fuzz testing for RESTful APIs. Through a rigorous comparison with Epsilon-Greedy and the application of targeted evaluation metrics, it was shown that such enhancements lead to faster

convergence, higher input validity, better exploitation-exploration balance and ultimately, more effective vulnerability detection.

The methodology developed throughout this research offers a structured and extensible foundation for intelligent fuzzing. From defining the exploration policy to instrumenting and visualizing key metrics, every aspect of the system was designed to align learning behavior with security objectives. Importantly, the inclusion of performance visualizations contributed not only to internal validation but also to the transparency and interpretability of the tool, an increasingly important consideration in security-focused AI applications.

Overall, this thesis contributes to the growing field of automated software testing by illustrating how modern reinforcement learning paradigms can be effectively applied to security testing tasks. While there remain challenges to scale and generalize the system, the results obtained support the viability and relevance of AI-driven fuzzers in enhancing software robustness in today's complex and dynamic API-driven environments.

Bibliography References

[1] OWASP, “API Security Top 10,” https://owasp.org/API-Security (accessed Apr. 16, 2025).

[2] M. Fielding, “Architectural Styles and the Design of Network-based Software Architectures,” https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm (accessed Apr. 16, 2025).

[3] Salt Security, “State of API Security Report 2023,” https://content.salt.security/rs/352-UXR-417/images/SaltSecurity-Report-State_of_API_Security.pdf (accessed Apr. 16, 2025).

[4] Salt Security, “Salt Security Perspective on the 2024” https://salt.security/blog/a-salt-security-perspective-on-the-2024-gartner-r-market-guide-for-api-protection (accessed Apr. 16, 2025).

[5] Bright, “What Is Fuzzing (Fuzz Testing)? Everything You Need to Know” https://www.brightsec.com/blog/fuzzing/ (accessed Apr. 16, 2025).

[6] H. Boehme et al., “Fuzzing: What Is It and How Does It Work?” https://security.berkeley.edu/resources/best-practices-how-articles/fuzzing-overview (accessed Apr. 16, 2025).

[7] Microsoft Research, “RESTler: Intelligent Fuzzing for REST APIs,” https://github.com/microsoft/restler-fuzzer (accessed Apr. 16, 2025).

[8] FuzzTheRest, “Automated REST API Fuzzing,” https://arxiv.org/abs/2407.14361 (accessed Apr. 16, 2025).

[9] G. Klees et al., “Evaluating Fuzz Testing,” https://dl.acm.org/doi/10.1145/3243734.3243804 (accessed Apr. 16, 2025).

[10] Science Direct, “The role of Reinforcement Learning in software testing” https://www.sciencedirect.com/science/article/pii/S0950584923001805 (accessed Apr. 16, 2025).

[11] D. Vengerov, “Adaptive coordination among fuzzy reinforcement learning agents performing distributed dynamic load balancing” https://ieeexplore.ieee.org/document/1004983 (accessed Apr. 16, 2025).

[12] Google Cloud, “Introduction to Microservices Architecture” https://cloud.google.com/architecture/microservices-architecture-introduction (accessed Apr. 16, 2025).

[13] OWASP, “Common API Security Issues,” https://owasp.org/www-project-api-security (accessed Apr. 16, 2025).

[14] M. Sutton et al., “Fuzzing: Brute Force Vulnerability Discovery,” https://dl.acm.org/doi/10.5555/1324770 (accessed Apr. 16, 2025).

[15] H. Amit, “Deep Learning vs Reinforcement Learning” https://medium.com/@heyamit10/deep-reinforcement-learning-vs-deep-learning-5e76763c7b4b (accessed Dec. 30, 2024).

[16] A. Abo-eleneen, “The role of Reinforcement Learning in software testing” https://www.sciencedirect.com/science/article/pii/S0950584923001805 (accessed Apr. 16, 2025).

[17] GECAD – Research Group on Intelligent Engineering and Computing for Advanced Innovation and Development, “About GECAD,” https://www.gecad.isep.ipp.pt/about/ (accessed Apr. 16, 2025).

[18] ISEP – Instituto Superior de Engenharia do Porto, “Research at ISEP,” https://www.isep.ipp.pt/Page/ViewPage/research (accessed Apr. 16, 2025).

[19] Tech Target, “What is Agile software development?” https://www.techtarget.com/searchsoftwarequality/definition/agile-software-development (accessed Apr. 16, 2025).

[20] Iscap, “IPP - Código de Boas Práticas e Conduta” https://www.iscap.ipp.pt/regulamentos/CodigoboaspraticasedecondutaIPP.pdf (accessed Apr. 16, 2025).

[21] IEEE, “IEEE Code of Ethics” https://www.ieee.org/about/corporate/governance/p7-8 (accessed Apr. 16, 2025).

[22] Dante Wu, “What is Automated Software Testing? - A Complete Guide.” https://www.headspin.io/blog/what-is-test-automation-a-comprehensive-guide-on-automated-testing (accessed Dec. 30, 2024).

[23] Sonar, “What Are Software Bugs? Definition Guide, Types & Tools | Sonar.” https://www.sonarsource.com/learn/software-bugs/ (accessed Dec. 30, 2024).

[24] Infosec, “Fuzzing: Mutation vs. generation | Infosec.” https://www.infosecinstitute.com/resources/hacking/fuzzing-mutation-vs-generation/

[25] Fuzzing Book, “Mutation-Based Fuzzing - The Fuzzing Book.” https://www.fuzzingbook.org/html/MutationFuzzer.html (accessed Dec. 30, 2024).

[26] GitLab, “Coverage-guided fuzz testing | GitLab.” https://docs.gitlab.com/ee/user/application_security/coverage_fuzzing/ (accessed Dec. 30, 2024).

[27] American Fuzzy Lop, “What is American Fuzzy Lop?” https://lcamtuf.coredump.cx/afl/ (accessed Dec. 30, 2024).

[28] LibFuzzer, “libFuzzer - a library for coverage-guided fuzz testing.” https://llvm.org/docs/LibFuzzer.html (accessed Dec. 30, 2024).

[29] International Business Machines Corporation, “What Is Artificial Intelligence (AI)? | IBM.” https://www.ibm.com/think/topics/artificial-intelligence (accessed Dec. 30, 2024).

[30] International Business Machines Corporation, “What Is Machine Learning (ML)? | IBM.” https://www.ibm.com/think/topics/machine-learning (accessed Dec. 30, 2024).

[31] Papers With Code, “Adaptive Input Representations Explained | Papers With Code.” https://paperswithcode.com/method/adaptive-input-representations (accessed Dec. 30, 2024).

[32] Scribbr, “What is the exploration vs exploitation trade off in reinforcement learning?” https://www.scribbr.com/frequently-asked-questions/what-is-the-exploration-vs-exploitation-trade-off-in-reinforcement-learning/ (accessed Dec. 30, 2024).

[33] Tan, S., Xu, X., & Wang, X., “Fuzz Testing: Art, Science and Engineering” https://ieeexplore.ieee.org/document/9513282 (accessed Apr. 27, 2025).

[34] Pricefx Knowledge Base, “How Input Generation Mode Works - Configuration Engineer Knowledge Base - Pricefx Knowledge Base.” https://knowledge.pricefx.com/space/KB/296091677/How+Input+Generation+Mode+Works (accessed Dec. 30, 2024).

[35] Russell, S., & Norvig, P., “Artificial Intelligence: A Modern Approach (4th ed.)” https://aima.cs.berkeley.edu/ (accessed Apr. 27, 2025).

[36] International Business Machines Corporation, “What Is Deep Learning? | IBM.” https://www.ibm.com/think/topics/deep-learning (accessed Dec. 30, 2024).

[37] Altexsoft, “Quality Assurance (QA), Quality Control and Testing.” https://www.altexsoft.com/whitepapers/quality-assurance-quality-control-and-testing-the-basics-of-software-quality-management/ (accessed Dec. 30, 2024).

[38] Mitchell, T. M., “Machine Learning. McGraw-Hill Education” https://www.cs.cmu.edu/~tom/mlbook.html (accessed Apr. 27, 2025).

[39] Goodfellow, I., Bengio, Y., & Courville, A., “Deep Learning. MIT Press” https://www.deeplearningbook.org/ (accessed Apr. 27, 2025).

[40] International Business Machines Corporation, “What is reinforcement learning?” https://www.ibm.com/think/topics/reinforcement-learning (accessed Dec. 30, 2024).

[41] Geeks for Geeks, “Epsilon-Greedy Algorithm in Reinforcement Learning” https://www.geeksforgeeks.org/epsilon-greedy-algorithm-in-reinforcement-learning/ (accessed Dec. 30, 2024).

[42] Medium, “Reinforcement Learning - Lesson 10: Exploration Strategies in Reinforcement Learning” https://medium.com/@nerdjock/reinforcement-learning-lesson-10-exploration-strategies-in-reinforcement-learning-f3e2010cec59 (accessed Dec. 30, 2024).

[43] Geeks for Geeks, “Upper Confidence Bound Algorithm in Reinforcement Learning” https://www.geeksforgeeks.org/upper-confidence-bound-algorithm-in-reinforcement-learning/ (accessed Dec. 30, 2024).

[44] University of Leicester, “What is PRISMA and why do you need a protocol?” https://le.ac.uk/library/research-support/systematic-reviews/prisma/ (accessed Dec. 30, 2024).

[45] Publish or Perish, “Publish or Perish” https://harzing.com/resources/publish-or-perish (accessed Dec. 30, 2024).

[46] Amin, MRRM, & Othman, MF (2021). Re-exploration of ε-greedy in deep reinforcement learning. RiTA 2020: Proceedings of the 8th …, Springer, https://doi.org/10.1007/978-981-16-4803-8_27 (accessed Dec. 30, 2024).

[47] D'Eramo, C, Cini, A, & Restelli, M (2019). Exploiting action-value uncertainty to drive exploration in reinforcement learning. 2019 International Joint …, ieeexplore.ieee.org, https://ieeexplore.ieee.org/abstract/document/8852326/ (accessed Dec. 30, 2024).

[48] Zangirolami, V, & Borrotti, M (2024). Dealing with uncertainty: Balancing exploration and exploitation in deep recurrent reinforcement learning. Knowledge-Based Systems, Elsevier, https://www.sciencedirect.com/science/article/pii/S0950705124002983 (accessed Dec. 30, 2024).

[49] Sharma, K, Singh, B, Herman, E, & ... (2021). Maximum information measure policies in reinforcement learning with deep energy-based model. 2021 International …, ieeexplore.ieee.org, https://ieeexplore.ieee.org/abstract/document/9410756/ (accessed Dec. 30, 2024).

[50] Sukhija, B, Coros, S, Krause, A, Abbeel, P, & ... (2024). MaxInfoRL: Boosting exploration in reinforcement learning through information gain maximization. arXiv preprint arXiv …, arxiv.org, https://arxiv.org/abs/2412.12098 (accessed Dec. 30, 2024).

[51] Thadikamalla, S, & Joshi, P (2023). Exploration Strategies in Adaptive Traffic Signal Control: A Comparative Analysis of Epsilon-Greedy, UCB, Softmax and Thomson Sampling. 2023 7th International Symposium on …, ieeexplore.ieee.org, https://ieeexplore.ieee.org/abstract/document/10391701/ (accessed Dec. 30, 2024).

[52] Gupta, H, Kong, ST, Srikant, R, & Wang, W (2019). Almost boltzmann exploration. arXiv preprint arXiv:1901.08708, arxiv.org, https://arxiv.org/abs/1901.08708 (accessed Dec. 30, 2024).

[53] Zhang, S, Li, S, Wu, F, & Li, XY (2023). Reinforcement Learning for Node Selection in Mixed Integer Programming. … Conference on Mobile Ad Hoc and …, ieeexplore.ieee.org, https://ieeexplore.ieee.org/abstract/document/10298331/ (accessed Dec. 30, 2024).

[54] Shani, L, Efroni, Y, & Mannor, S (2019). Exploration conscious reinforcement learning revisited. … on machine learning, proceedings.mlr.press, https://proceedings.mlr.press/v97/shani19a.html (accessed Dec. 30, 2024).

[55] Tiwari, PK, Singh, P, Rajagopal, NK, & ... (2023). IoT-Based Reinforcement Learning Using Probabilistic Model for Determining Extensive Exploration through Computational Intelligence for Next-Generation …. Computational …, Wiley Online Library, https://doi.org/10.1155/2023/5113417 (accessed Dec. 30, 2024).

[56] Li, M, Huang, T, & Zhu, W (2021). Adaptive exploration policy for exploration–exploitation tradeoff in continuous action control optimization. International Journal of Machine Learning and …, Springer, https://doi.org/10.1007/s13042-021-01387-5 (accessed Dec. 30, 2024).

[57] Medium, "What is Exploration Strategies in Reinforcement Learning?" https://medium.com/@aiblogtech/what-is-exploration-strategies-in-reinforcement-learning-32677239245e (accessed Dec. 30, 2024).

[58] Geeks for Geeks, "Q-Learning in Reinforcement Learning" https://www.geeksforgeeks.org/q-learning-in-python/ (accessed Apr. 27, 2025).

[59] Automatic Addison, "Boltzmann Distribution and Epsilon Greedy Search" https://automaticaddison.com/boltzmann-distribution-and-epsilon-greedy-search/ (accessed Apr. 27, 2025).

[60] David G., Jobst H., Wolfram B., "Deterministic Model of Incremental Multi-Agent Boltzmann Q-Learning: Transient Cooperation, Metastability and Oscillations" https://arxiv.org/abs/2501.00160 (accessed Apr. 27, 2025).

[61] Luca D., Hugh Z., Marc L., David C., "Easy as ABCs: Unifying Boltzmann Q-Learning and Counterfactual Regret Minimization" https://arxiv.org/abs/2402.11835 (accessed Apr. 27, 2025).

[62] Regulations.gov, "Regulations.gov API Documentation" https://open.gsa.gov/api/regulationsgov/ (accessed Apr. 27, 2025).

[63] Dados.gov.pt, "Dados.gov.pt – Portuguese Open Data Portal" https://dados.gov.pt/pt/ (accessed Apr. 27, 2025).

[64] Data Usa, "Data USA API Documentation" https://datausa.io/about/api/ (accessed Apr. 27, 2025).

[65] OpenFDA, “OpenFDA API Overview” https://open.fda.gov/apis/ (accessed Apr. 27, 2025).

[66] Fiscal Data, “Fiscal Data Treasury API” https://fiscaldata.treasury.gov/api-documentation/#how-to-access-our-api (accessed Apr. 27, 2025).

[67] NASA, “NASA Open APIs” https://api.nasa.gov/ (accessed Apr. 27, 2025).

[68] Github, “GitHub REST API Documentation” https://docs.github.com/en/rest (accessed Apr. 27, 2025).

[69] Gitlab, “GitLab API Reference” https://docs.gitlab.com/ee/api/ (accessed Apr. 27, 2025).

[70] Jenkins, “Jenkins Remote Access API” https://www.jenkins.io/doc/book/using/remote-access-api/ (accessed Apr. 27, 2025).

[71] Docker, “Docker Hub API v2” https://docs.docker.com/docker-hub/api/latest/ (accessed Apr. 27, 2025).

[72] Kubernetes, “Kubernetes API Reference” https://kubernetes.io/docs/reference/kubernetes-api/ (accessed Apr. 27, 2025).

[73] Terraform, “Terraform Cloud API Docs” https://www.terraform.io/cloud-docs/api-docs (accessed Apr. 27, 2025).

[74] Sentry, “Sentry API Documentation” https://docs.sentry.io/api/ (accessed Apr. 27, 2025).

[75] SonarQube, “SonarQube Web API” https://docs.sonarsource.com/sonarqube-server/latest/extension-guide/web-api/ (accessed Apr. 27, 2025).

[76] Jira, “Jira Cloud REST API v3” https://developer.atlassian.com/cloud/jira/platform/rest/v3/ (accessed Apr. 27, 2025).

[77] Bitbucket, “Bitbucket API v2” https://developer.atlassian.com/bitbucket/api/2/reference/resource/ (accessed Apr. 27, 2025).

[78] AWS, “AWS CodeBuild API Reference” https://docs.aws.amazon.com/codebuild/latest/APIReference/Welcome.html (accessed Apr. 27, 2025).

[79] Azure, “Azure DevOps REST API Reference” https://learn.microsoft.com/en-us/rest/api/azure/devops/?view=azure-devops-rest-7.1 (accessed Apr. 27, 2025).

[80] Cristian L., “Reinforcement Learning 101: Q-Learning” https://towardsdatascience.com/reinforcement-learning-101-q-learning-27add4c8536d/ (accessed Apr. 27, 2025).

[81] Fumihiko I., Takahiro S., Yutaka S., Hiroyuki S., “Reinforcement-learning agents with different temperature parameters explain the variety of human action–selection behavior in a Markov decision process task” https://www.sciencedirect.com/science/article/abs/pii/S0925231208002518 (accessed Apr. 27, 2025).

[82] Ashutosh M., “Reinforcement Learning 6: Exploration vs Exploitation” https://ashutoshmakone.medium.com/reinforcement-learning-5-exploration-vs-exploitation-c1bae5a2ea42 (accessed Apr. 27, 2025).

[83] Alkan E., “Intro to Q-Learning” https://www.kaggle.com/code/alkanerturan/intro-to-q-learning (accessed Apr. 27, 2025).

[84] Valentina Z., Matteo B., “Dealing with uncertainty: Balancing exploration and exploitation in deep recurrent reinforcement learning” https://www.sciencedirect.com/science/article/pii/S0950705124002983 (accessed Apr. 27, 2025).

[85] Everton G., “Optimizing Decision-Making in AI: Fine-Tuning the Epsilon-Greedy Algorithm for Enhanced Performance” https://medium.com/operations-research-bit/optimizing-decision-making-in-ai-fine-tuning-the-epsilon-greedy-algorithm-for-enhanced-performance-ea61e86d6f1d (accessed Apr. 27, 2025).