INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

MESTRADO EM ENGENHARIA DE INTELIGÊNCIA ARTIFICIAL

# FuzzTheREST - Intelligent Automated Black-box RESTful API Fuzzer

TIAGO FONTES DIAS

setembro de 2023

INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

MESTRADO EM ENGENHARIA DE INTELIGÊNCIA ARTIFICIAL

# FuzzTheREST - Intelligent Automated Black-box RESTful API Fuzzer

TIAGO FONTES DIAS

Setembro de 2023

# FuzzTheREST - Intelligent Automated Black-box RESTful API Fuzzer

**Tiago Fontes Dias**

**1180939**

**Dissertation to obtain the Master's Degree in Artificial Intelligence Engineering**

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

**Co-supervisor: Dr. Eva Catarina Gomes Maia**

**Jury**:

President:

Dr. Ana Almeida, Coordinator Professor, Polytechnic of Porto

Vogais:

Dr. Mário Antunes, Coordinator Professor, Polytechnic of Leiria

Dr. Isabel Praça, Coordinator Professor, Polytechnic of Porto

Porto, September 2023

*«To my family for all the love and support they have given me»*

# Abstract

In recent years, the pervasive influence of technology has deeply intertwined with human life, impacting diverse fields. This relationship has evolved into a dependency, with software systems playing a pivotal role, necessitating a high level of trust. Today, a substantial portion of software is accessed through Application Programming Interfaces, particularly web APIs, which predominantly adhere to the Representational State Transfer architecture. However, this architectural choice introduces a wide range of potential vulnerabilities, which are available and accessible at a network level. The significance of Software testing becomes evident when considering the widespread use of software in various daily tasks that impact personal safety and security, making the identification and assessment of faulty software of paramount importance.

In this thesis, FuzzTheREST, a black-box RESTful API fuzzy testing framework, is introduced with the primary aim of addressing the challenges associated with understanding the context of each system under test and conducting comprehensive automated testing using diverse inputs. Operating from a black-box perspective, this fuzzer leverages Reinforcement Learning to efficiently uncover vulnerabilities in RESTful APIs by optimizing input values and combinations, relying on mutation methods for input exploration. The system's value is further enhanced through the provision of a thoroughly documented vulnerability discovery process for the user. This proposal stands out for its emphasis on explainability and the application of RL to learn the context of each API, thus eliminating the necessity for source code knowledge and expediting the testing process. The developed solution adheres rigorously to software engineering best practices and incorporates a novel Reinforcement Learning algorithm, comprising a customized environment for API Fuzzy Testing and a Multi-table Q-Learning Agent.

The quality and applicability of the tool developed are also assessed, relying on the results achieved on two case studies, involving the Petstore API and an Emotion Detection module which was part of the CyberFactory#1 European research project. The results demonstrate the tool's effectiveness in discovering vulnerabilities, having found 7 different vulnerabilities and the agents' ability to learn different API contexts relying on API responses while maintaining reasonable code coverage levels.

**Keywords**: Automated Software Testing; Reinforcement Learning; Fuzzy Testing; Software Quality; RESTful APIs

# Resumo

Ultimamente, a influência da tecnologia espalhou-se pela vida humana de uma forma abrangente, afetando uma grande diversidade dos seus aspetos. Com a evolução tecnológica esta acabou por se tornar uma dependência. Os sistemas de software começam assim a desempenhar um papel crucial, o que em contrapartida obriga a um elevado grau de confiança. Atualmente, uma parte substancial do software é implementada em formato de Web APIs, que na sua maioria seguem a arquitetura de transferência de estado representacional. No entanto, esta introduz uma série vulnerabilidade. A importância dos testes de software torna-se evidente quando consideramos o amplo uso de software em várias tarefas diárias que afetam a segurança, elevando ainda mais a importância da identificação e mitigação de falhas de software.

Nesta tese é apresentado o FuzzTheREST, uma framework de teste fuzzy de APIs RESTful num modelo caixa preta, com o objetivo principal de abordar os desafios relacionados com a compreensão do contexto de cada sistema sob teste e a realização de testes automatizados usando uma variedade de possíveis valores. Este fuzzer utiliza aprendizagem por reforço de forma a compreender o contexto da API que está sob teste de forma a guiar a geração de valores de teste, recorrendo a métodos de mutação, para descobrir vulnerabilidades nas mesmas. Todo o processo desempenhado pelo sistema é devidamente documentado para que o utilizador possa tomar ações mediante os resultados obtidos. Esta explicabilidade e aplicação de inteligência artificial para aprender o contexto de cada API, eliminando a necessidade de analisar código fonte e acelerando o processo de testagem, enaltece e distingue a solução proposta de outras. A solução desenvolvida adere estritamente às melhores práticas de engenharia de software e inclui um novo algoritmo de aprendizagem por reforço, que compreende um ambiente personalizado para testagem Fuzzy de APIs e um Agente de Q-Learning com múltiplas Q-tables.

A qualidade e aplicabilidade da ferramenta desenvolvida também são avaliadas com base nos resultados obtidos em dois casos de estudo, que envolvem a conhecida API Petstore e um módulo de Deteção de Emoções que fez parte do projeto de investigação europeu CyberFactory#1. Os resultados demonstram a eficácia da ferramenta na descoberta de vulnerabilidades, tendo identificado 7 vulnerabilidades distintas, e a capacidade dos agentes em aprender diferentes contextos de API com base nas respostas da mesma, mantendo níveis de cobertura aceitáveis.

**Palavras-chave**: Testagem de Software Automatizada; Aprendizagem por Reforço; Testagem Fuzzy; Qualidade de Software; APIs RESTful

# Acknowledgements

I would like to express my sincere gratitude to everyone involved in the success of the work presented in this thesis.

Firstly, I would like to thank my supervisor, Prof.ª Dra. Isabel Praça and co-supervisor, Dra. Eva Maia, for their guidance and mentorship throughout this research work, and for all the lessons taught the past two years we've worked together. Their expertise in the research field has been instrumental in shaping the quality of my works. I´m deeply thankful for all the time and effort they've invested in me.

I would also like to thank my mates, Shaq, and Vito for all their support and companionship, for all the great moments spent together working, reflecting on life, joking and extracurricular activities. You know what I'm talking about guys… On a more serious note, I could not have asked for better colleagues, their work method is incredible and so is the outcome of working with them.

I would also like to thank my GECAD co-workers, who have accompanied me in this journey and had to put up with my uncertainties and doubts throughout this work, for all the joyful moments and talks we had about work and life in general. Their presence had undoubtedly positively impacted my work.

I extend my heartfelt thanks to all MEIA professors for all their teaching and for passing me great knowledge in the fields of AI.

I'd also like to thank my all my friends, who've always trusted and continue to trust in my capacities and motivate me to strive for improvement. Their involvement in various aspects of my life, such as our nights out, shared music interests, shared hobbies bring me immense joy and makes me appreciate life a bit more.

Last, but definitely not least, I would like to thank my life pillars; My parents Alexandra and António for all their love, encouragement, for believing in my capacities, and for all the sacrifices they've made to provide me with the best upbringing and education possible; My brothers André and Martim for all always tolerating my bad moods and for helping me stay on the right track, ensuring I don't get too sidetracked by work and other distractions; The rest of my family for always being there for me. Oh, and of course, Cookie, my trusty 22cm night's watch!

# Index

**1 Introduction ..... 1**

1.1 Context ..... 1

1.2 Problem Statement ..... 3

1.3 Research Questions and Objectives ..... 4

1.4 Scientific Contributions ..... 5

1.5 Outline ..... 6

**2 State of the Art ..... 7**

2.1 Research Method ..... 7

2.2 Automated Software Testing ..... 9
- 2.2.1 Principles and Levels of Testing ..... 11
- 2.2.2 Black-box Testing ..... 13
- 2.2.3 Fuzzy Testing ..... 18

2.3 Artificial Intelligence ..... 21
- 2.3.1 Reinforcement Learning ..... 22

2.4 Resume ..... 27

**3 FuzzTheREST ..... 29**

3.1 Conceptualisation ..... 29
- 3.1.1 Target Audience ..... 30
- 3.1.2 Ethical and Security Considerations ..... 30

3.2 Software Engineering ..... 31
- 3.2.1 Requirements Engineering ..... 32
- 3.2.2 Practical Software Engineering Principles ..... 33
- 3.2.3 Architectural Design ..... 34

3.3 FuzzCore: Reinforcement Learning Design ..... 39
- 3.3.1 Data Acquisition and Processing ..... 40
- 3.3.2 Reinforcement Learning Environment ..... 42
- 3.3.3 Multi-table Q-Learning Agent ..... 44

3.4 Resume ..... 46

**4 Demonstration ..... 47**

4.1 Petstore API Case Study ..... 48
- 4.1.1 Execution and Results ..... 50

4.2 CyberFactory#1 Case Study ..... 55
- 4.2.1 Execution and Results ..... 56

4.3 Discussion of Results ..... 58

4.4 Resume ..... 59

**5 Conclusion ........ 61**

5.1 Summary of results ........ 61

5.2 Objectives Achieved ........ 62

5.3 Research Questions Answered ........ 62

5.3.1 RQ1 - What are the main flaws of current automated Software testing applications? ........ 63

5.3.2 RQ2 – Which specifications/methods can be utilised to avoid exhaustive web API testing? ........ 63

5.3.3 RQ3 – Which are the methods utilised for test/input generation? ........ 63

5.3.4 RQ4 – Is Reinforcement Learning an alternative solution to what would be considered a search-based problem? ........ 64

5.4 Limitations and Future Work ........ 64

**Appendixes ........ 77**

Appendix A – Process View ........ 78

Appendix B- Agent Training Process View ........ 79

Appendix C – Petstore API Result Plots ........ 80

C.1 – AddPet ........ 80

C.2 – UpdatePet ........ 82

C.3 – GetPetById ........ 84

C.4 – FindPetsByStatus ........ 86

C.5 – FindPetsByTags ........ 88

C.6 – UploadFile ........ 90

C.7 – UpdatePetWithForm ........ 92

C.8 – DeletePet ........ 94

C.9 – CreateUser ........ 96

C.10 – LoginUser ........ 98

C.11 – UpdateUser ........ 100

C.12 – LogoutUser ........ 102

C.13 – GetUserByName ........ 104

C.14 – DeleteUser ........ 106

C.15 – PlaceOrder ........ 108

C.16 – GetOrderById ........ 110

C.17 – GetInventory ........ 112

C.18 – DeleteOrder ........ 114

Appendix D – Status Codes Function Distribution ........ 116

# List of Figures

Figure 1 – Software Testing Workflow .......... 2
Figure 2 – Search Query .......... 8
Figure 3 - PRISMA Diagram .......... 9
Figure 4 – A model of the Software testing process [2] .......... 10
Figure 5 – Equivalence Partitioning and Boundary-value Examples [2] .......... 14
Figure 6 – Fuzzy Testing Flowchart .......... 18
Figure 7 – Artificial Intelligence Fields .......... 21
Figure 8 – RL Algorithms Flowchart [70] .......... 23
Figure 9 – Reinforcement Learning algorithms Taxonomy .......... 24
Figure 10 – Deep Q-Network Architecture [80] .......... 26
Figure 11 - 4+1 View Model [93] .......... 34
Figure 12 - C4 Models and 4+1 Architectural View Crossover Matrix .......... 35
Figure 13 – Level 2 logic View of FuzzTheREST .......... 36
Figure 14 – Level 3 Logic View of FuzzCore .......... 37
Figure 15 – Petstore API OAS file [109] .......... 40
Figure 16 - OAS File Data Extraction .......... 41
Figure 17 - FuzzTheREST Data Schema .......... 41
Figure 18 – Reinforcement Learning Environment and Agent Overview .......... 43
Figure 19 - Q-Value Convergence of addPet Function .......... 51
Figure 20 - Total HTTP Code Status Received .......... 51
Figure 21 - JaCoCo's Code Coverage Report .......... 54
Figure 22 – Human Behaviour Analyzer architecture [9] .......... 55
Figure 23 – DetectEmotion Agent’s Q-Value Convergence .......... 56
Figure 24 - Randomly Generated Test Frame .......... 57
Figure 25 - DetectEmotion Agent’s State Visits .......... 57
Figure 26 – Level 3 Process View .......... 78
Figure 27 – Agent Training Process View .......... 79
Figure 28 – AddPet Agent’s Action Distribution .......... 80
Figure 29 - AddPet Agent’s Q-Value Convergence and State Visits .......... 81
Figure 30 - UpdatePet Agent’s Action Distribution .......... 82
Figure 31 - UpdatePet Agent’s Q-Value Convergence and State Visits .......... 83
Figure 32 - GetPetById Agent’s Action Distribution .......... 84
Figure 33 - GetPetById Agent’s Q-Value Convergence and State Visits .......... 85
Figure 34 - FindPetsByStatus Agent’s Action Distribution .......... 86
Figure 35 - FindPetsByStatus Agent’s Q-Value Convergence and State Visits .......... 87
Figure 36 - FindPetsByTags Agent’s Action Distribution .......... 88
Figure 37 – FindPetsByTags Agent’s Q-Value Convergence and State Visits .......... 89
Figure 38 - UploadFile Agent’s Action Distribution .......... 90
Figure 39 - UploadFile Agent’s Q-Value Convergence and State Visits .......... 91
Figure 40 - UpdatePetWithForm Agent’s Action Distribution .......... 92

Figure 41 - UpdatePetWithForm Agent’s Q-Value Convergence and State Visits .................... 93
Figure 42 - DeletePet Agent’s Action Distribution .................................................................. 94
Figure 43 - DeletePet Agent’s Q-Value Convergence and State Visits ..................................... 95
Figure 44 - CreateUser Agent’s Action Distribution ................................................................. 96
Figure 45 - CreateUser Agent’s Q-Value Convergence and State Visits ................................... 97
Figure 46 - LoginUser Agent’s Action Distribution ................................................................... 98
Figure 47 - LoginUser Agent’s Q-Value Convergence and State Visits ..................................... 99
Figure 48 - UpdateUser Agent’s Action Distribution ............................................................... 100
Figure 49 - UpdateUser Agent’s Q-Value Convergence and State Visits ................................ 101
Figure 50 - LogoutUser Agent’s Action Distribution ............................................................... 102
Figure 51 – LogoutUser Agent’s Q-Value Convergence and State Visits ................................ 103
Figure 52 - GetUserByName Agent’s Action Distribution ....................................................... 104
Figure 53 – GetUserByName Agent’s Q-Value Convergence and State Visits ........................ 105
Figure 54 - DeleteUser Agent’s Action Distribution ................................................................ 106
Figure 55 - DeleteUser Agent’s Q-Value Convergence and State Visits ................................. 107
Figure 56 - PlaceOrder Agent’s Action Distribution ................................................................ 108
Figure 57 – PlaceOrder Agent’s Q-Value Convergence and State Visits ................................ 109
Figure 58 - GetOrderById Agent’s Action Distribution ............................................................ 110
Figure 59 - GetOrderById Agent’s Q-Value Convergence and State Visits ............................. 111
Figure 60 – GetInventory Agent’s Action Distribution ............................................................ 112
Figure 61 - GetInventory Agent’s Q-Value Convergence and State Visits .............................. 113
Figure 62 - DeleteOrder Agent’s Action Distribution .............................................................. 114
Figure 63 - DeleteOrder Agent’s Q-Value Convergence and State Visit ................................. 115
Figure 64 - Functions Contributions to 2XX Status Code ........................................................ 116
Figure 65 - Functions Contributions to 4XX Status Code ........................................................ 116
Figure 66 - Functions Contributions to 5XX Status Code ........................................................ 117

# List of Tables

Table 1 – Inclusion and Exclusion Criteria ........ 8
Table 2 – Methods for black-box Software testing ........ 15
Table 3 – Level 3 Fuzzer components responsibilities ........ 38
Table 4 – FuzzTheREST data generation/mutation methods [60], [110] ........ 42
Table 5 – HTTP status codes description [111] ........ 43
Table 6 - Case Study Algorithm Parameters ........ 48
Table 7 - Petstore API Pets Methods ........ 49
Table 8 - Petstore API Store Methods ........ 49
Table 9 - Petstore API Users Methods ........ 50
Table 10 - Petstore Vulnerabilities ........ 52
Table 11 – Objectives fulfilled ........ 62

# Acronyms

## Acronyms

| | |
|---|---|
| **ACM DL** | *Association for Computing Machinery Digital Library* |
| **AFL** | *American Fuzzy Lop* |
| **AI** | *Artificial Intelligence* |
| **API** | *Application Programming Interfaces* |
| **AST** | *Automated Software Testing* |
| **AUTH** | *Authentication* |
| **BMP** | *Basic Multilingual Plane* |
| **CRUD** | *Create, Read, Update and Delete* |
| **CSV** | *Comma Separated Values* |
| **DFA** | *Deterministic Finite Automaton* |
| **DQN** | *Deep Q-Network* |
| **EC** | *Exclusion Criteria* |
| **FoF** | *Factories of the Future* |
| **GA** | *Genetic Algorithm* |
| **GRASP** | *General Responsibility Assignment Software Patterns* |
| **HBA** | *Human Behaviour Analyzer* |
| **HTML** | *HyperText Markup Language* |
| **HTTP** | *Hypertext Transfer Protocol* |
| **IC** | *Inclusion Criteria* |
| **IEEE** | *Institute of Electrical and Electronics Engineers* |
| **IT** | *Information Technology* |
| **IoT** | *Internet of Things* |

| | |
|---|---|
| **JSON** | *JavaScript Object Notation* |
| **JVM** | *Java Virtual Machine* |
| **MCC** | *Monte Carlo Control* |
| **MDP** | *Markov Decision Processes* |
| **MDPI** | *Multidisciplinary Digital Publishing Institute* |
| **ML** | *Machine Learning* |
| **NIST** | *National Institute of Standards and Technology* |
| **O** | *Objective* |
| **OAS** | *OpenAPI Specification* |
| **OOD** | *Object-Oriented Design* |
| **OS** | *Operating System* |
| **PPO** | *Proximal Policy Optimization* |
| **PRISMA** | *Preferred Reporting Items for Systematic Reviews and Meta-Analyses* |
| **REST** | *Representational State Transfer* |
| **RL** | *Reinforcement Learning* |
| **ROI** | *Return Of Investment* |
| **RQ** | *Research Question* |
| **SARSA** | *State-Action-Reward-State-Action* |
| **SQL** | *Structured Query Language* |
| **SUT** | *Software Under Test* |
| **UI** | *User Interface* |
| **UML** | *Unified Modeling Language* |
| **URL** | *Uniform Resource Locator* |
| **US** | *United States* |
| **V&V** | *Verification and Validation* |
| **XML** | *eXtensible Markup Language* |

## Symbols

| | |
|---|---|
| ***δ*** | *Likelyhood of Transictioning States* |
| **π*** | *Optimal Policy* |
| ***S/s*** | *Finite Number of States* |
| ***A/a*** | *Finite number of Actions* |
| ***T*/t** | *Transaction* |
| ***R*** | *Reward* |
| $\alpha$ | *Learning Rate* |
| $\gamma$ | *Discount Factor* |
| $\mathbf{Q}$ | *Q-values* |

# 1 Introduction

This chapter consists of a description of the problem addressed in this thesis along with the motivations to do so, whilst providing contextual information on the topic. Research questions, objectives and contributions are also described along with the document's outline.

## 1.1 Context

Over the years, humans have found a relationship with technology. Technology has affected human lives in the most various fields, such as communication, education, medicine, transportation, and work, amongst others. However, this relationship has become somewhat of a dependency. Software systems have become so crucial from a professional and personal standpoint that human interaction is unavoidable. Consequently, people have to fully trust these systems. The trustiness of Software directly relates to its quality [1].

Software Testing is the process that assesses and ensures Software quality. It involves evaluating a Software application or system to determine if it works correctly and meets the specified requirements [1]. Typically, it consists of executing the Software Under Test (SUT) and monitoring it for errors, bugs, and other issues. It is usually performed by specialised teams or organisations that are dedicated to testing Software applications and systems. These teams typically use a variety of tools and techniques to test the Software, including automated testing tools, management tools, and testing frameworks. The goal of Software testing is to identify any problems or defects in the Software so that they can be addressed and resolved, preferably before its release, to ensure its success and increase longevity [1]. This is extremely important to guarantee the Software's quality and discourage malicious users of exploiting these systems, which can be extremely disruptive to other person's lives.

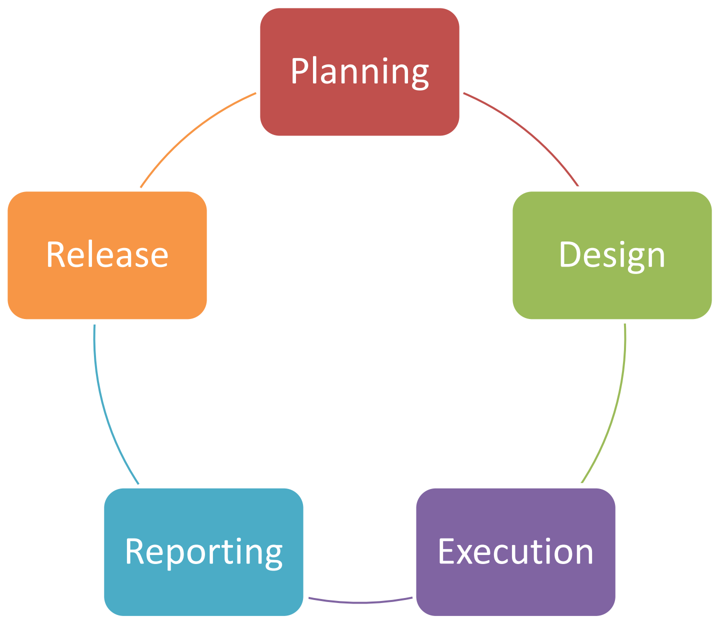


Figure 1 – Software Testing Workflow

Considering the Software development process, Software testing is typically performed at various stages, starting with individual units or components of the Software and progressing to larger, more complex systems. This helps ensure the correctness of the Software whilst meeting the specified requirements at every stage of the development process. As shown in Figure 1, it can be divided into five distinct steps, with some stages overlapping or being performed iteratively depending on the specific needs and goals of the testing effort [2]:

- Planning: This is the first stage of Software testing and involves defining the scope, objectives, and approach for the testing effort;
- Design: In this phase, test cases and scenarios are designed and developed to test the Software;
- Execution: After producing the test cases and scenarios, these are executed, and their results are recorded;
- Reporting: This stage involves documenting and reporting the results of the testing, including any errors, bugs, or other issues that were discovered;
- Release: Once the testing is complete and any necessary fixes have been made, the Software is ready for release. This stage typically involves performing final checks and quality assurance processes to ensure that the Software is ready for use.

Nowadays, most Software is available via Application Programming Interfaces (APIs) for its ability to provide web services in a lightweight, maintainable, and scalable way. These APIs can be of many types, but the most common are web APIs [3]. These usually follow a Representational State Transfer (REST) API architecture, and their communication occurs in the network via Hypertext Transfer Protocol (HTTP) requests. This architecture opens more opportunities to exploit the backend systems, ranging from security vulnerabilities to data integrity, performance, and compatibility issues.

GECAD [4] is an R&D unit having as its mission the development of scientific research and innovation for incorporating intelligence in engineering and computing complex systems. GECAD research areas like Affective Computing, Ambient Intelligence, Artificial Intelligence, Cyber-physical Systems, Group Decision Support, and the Internet of Things (IoT). Energy is the main field of application in this R&D area. However, other areas such as Industry, Cities, Security and Tourism are also considered. GECAD has proposed the project tackled by this thesis within the scope of “Cyber SeC IP”.

This thesis tackles the quality of Software, specifically web APIs, by introducing an automated tool for Software testing in a black-box setting, meaning that it has no knowledge nor access to the system's inner workings. This test setting aims to identify Software defects that white-box testing performed by developers may not have caught, resorting to fuzzy testing. However, being black-box, the input values search and combinatorial spaces are enormous. To reduce the complexity of this search, Artificial Intelligence (AI) methods such as metaheuristics are capable of exploring this search space with the goal of finding the right input values and combinations. This thesis considers the use of Reinforcement Learning to help solve this problem by getting to know the environment and take better actions to obtain the faulty input necessary to find vulnerabilities. Therefore, this tools's goal is to automate black-box testing by generating test cases resorting to Reinforcement Learning (RL) to solve the search and combinatorial problem that is generating and combining input parameters to interact with functionalities provided by a web API. Parameterised via configuration files that are both automatically and manually generated, the objective is also to cover the more hard-to-reach paths.

Tackling a project of this dimension, with such high learning curve, at the forefront of the future of technology is no easy task. However, it greatly increases one's research and engineering skills.

## 1.2 Problem Statement

A Software can be evaluated by its quality. Higher Software quality usually leads to more correct and secure systems, which reflects on the confidence with which users interact with it. Software testing is one of the main factors in increasing Software quality. However, this step is often rushed or even skipped, because it takes a lot of time and costs a lot of money. Nonetheless, the lack of testing has also shown to be quite expensive [5].

As determined by Consortium for Information and Software quality [6], the total cost of poor Software quality in the United States (US) was around 2 trillion dollars. The identification and revision of Software vulnerabilities ahead of its release helps avoid problems that could lead to user dissatisfaction, financial losses, or even legal issues. As reported by the National Institute of Standards and Technology (NIST), an average bug found in early development stages takes approximately 5 hours to be fixed, whereas, in a post-product release, it takes around 15.3 hours. Therefore, in addition to improving the quality of the Software, testing decreases development time and costs. Additionally, release delays can be avoided if issues are identified and addressed early in the development stage.

Moreover, with Software development quickly increasing and becoming more complex over the years, automated Software testing tools have been developed to assist testers in detecting possible vulnerabilities by executing automatically generated test cases on Software applications. One of the key benefits of automated testing is that it saves time and resources by automating many of the tasks that would otherwise need to be performed manually. As stated in [7], 44% of Information Technology (IT) companies automated 50% of testing in the year of 2020, with 24% seeing an increase in Return Of Investment (ROI). Although the test suites may not always be reliable, as there is a certain randomness associated with the generated test cases, they have proven to be quite helpful in uncovering previously unknown Software vulnerabilities often missed by testers [1].

Despite the integration of automated Software testing tools as part of the Software testing process, during the first quarter of 2022, over 8000 vulnerabilities were discovered and documented [8]. Especially for internet-facing applications, [8] found that one in ten vulnerabilities is considered of high or critical risk.

In retrospect, the failures in Software testing, described by the numbers, are scary since Software is used for diverse daily tasks that may include the safety or security of people. The existence of vulnerabilities in these systems can motivate hackers to exploit them and interfere with personal lives. Moreover, the lack of correctness discourages the use of Software or, when undetected, may even produce catastrophic results depending on the severity of the defect. As such, there is an urge to uncover and assess these and other unknown Software vulnerabilities.

## 1.3 Research Questions and Objectives

This thesis presents the work developed to address the automation of Software testing for Software defect discovery of RESTful APIs. To serve as guidance for this work, four Research Questions (RQ) were formulated. These are as follows:

- **RQ1 –** What are the main flaws of current automated Software testing applications?
- **RQ2 –** Which specifications/methods can be utilised to avoid exhaustive web API testing?
- **RQ3 –** Which are the methods utilised for test/input generation?
- **RQ4 –** Is Reinforcement Learning an alternative solution to what would be considered a search-based problem?

The main objective of this dissertation is the development of an automated Software testing tool that uses RL for test case generation to test RESTful APIs. Nonetheless, it also outputs other Objectives (O) ranging from the state of the art to design, implementation, and vulnerability discovery. These can be defined as follows:

- **O1 –** Investigate the current state of the art of automated Software testing;
- **O2 –** Investigate the current state of the art of constrained input generation;

- **O3 –** Identify the current flaws of automated Software testing tools and proposed solution;
- **O4 –** Propose a RL environment in the context of Software testing;
- **O5 –** Design and implement a RL-based automated Software testing tool equipped with methods for fuzzy black-box testing;
- **O6 –** In a case study, evaluate the developed work in a well-known public API and in a private API which integrates a GECAD project for vulnerability discovery.

## 1.4 Scientific Contributions

The completion of the previously established objectives presents multiple scientific Contributions (C):

- **C1 –** A survey about automated Software testing;
- **C2 –** An analysis of the methods to achieve automated black-box testing of web APIs;
- **C3 –** A survey about input/test generation;
- **C4 –** The definition of a RL approach in the Software testing field;
- **C5 –** The benefits of interpreting a search-based problem as a RL problem;
- **C6 –** A functional RL-based automated Software black-box testing tool;
- **C7 –** Application of the tool in two real-world case study.

Besides the scientific contributions presented by the completion of the referred objectives, throughout the development of this work, a total of four related scientific papers were published:

- [Conference] **Dias, T**., Maia, E., Praça, I. (2023). FuzzTheREST: Intelligent Automated Black-box RESTful API Fuzzer, to be submitted to International Symposium on Foundations & Practice of Security 2023 conference.
- [Conference] - **Dias, T.**, Batista, A., Maia, E., Praça, I. (2023). TestLab: An Intelligent Automated Software Testing Framework. In: Mehmood, R., *et al.* Distributed Computing and Artificial Intelligence, Special Sessions I, 20th International Conference. DCAI 2023. Lecture Notes in Networks and Systems, vol 741. Springer, Cham. https://doi.org/10.1007/978-3-031-38318-2_35 [12]
- [Conference] **Dias, T.**, Maia, E., Praça, I. (2023). RESTful API Automated Software Testing: A Systematic Review, to be submitted to Computers and Security Journal.
- [Journal – Under review] - Vitorino, J., **Dias, T.**, Fonseca, T., Maia, E., & Praça, I. (2023). Constrained Adversarial Learning and its applicability to Automated Software Testing: a systematic review. https://doi.org/10.48550/arXiv.2303.07546 [11]. Submitted to Information and Software Technology Journal, currently under review.
- [Conference] - Wannous, S., **Dias, T.**, Maia, E., Praça, I., Faria, A.R. (2022). Multiple Domain Security Awareness for Factories of the Future. In: González-Briones, A., et

al. Highlights in Practical Applications of Agents, Multi-Agent Systems, and Complex Systems Simulation. The PAAMS Collection. PAAMS 2022. Communications in Computer and Information Science, vol 1678. Springer, Cham. https://doi.org/10.1007/978-3-031-18697-4_3 [9]

- [Journal] - Maia, E.; Wannous, S.; **Dias, T.**; Praça, I.; Faria, A. Holistic Security and Safety for Factories of the Future. *Sensors* **2022**, *22*, 9915. https://doi.org/10.3390/s22249915 [10]
- [Conference] - **Dias T.**, Vitorino, J., Fonseca, T., Maia, E., Praça, I., Viamonte, M. (2023). Unravelling Network-based Intrusion Detection: A Neutrosophic Rule Mining and Optimization Framework, submitted to European Symposium on Research in Computer Security 2023 conference, to be presented in conference and indexed.

## 1.5 Outline

This thesis is divided into multiple chapters, which can be described as follows:

- Chapter 1 describes the problem statement that motivates this work from a social and technological viewpoint. The research questions and objectives that guide this work are also presented, as well as its scientific contributions.
- Chapter 2 consists of a state of the art on Software testing by describing theoretical key points such as Software quality and levels of testing and explores current black-box methods for detecting Software vulnerabilities. Additionally, it explores automated Software testing workflow and respective academic advances to achieve automation. Moreover, RL methods and techniques are also researched.
- Chapter 3 describes the proposed solution, by conceptualizing the tool, identifying its target audience, and tackling ethical and security considerations. The system architecture is also illustrated, following good Software Engineering practices, which are also presented. Lastly, the implementation of the RL components, environment, and agent, are also described.
- Chapter 4 describes the experimentation done to investigate the quality of the tool. In this chapter, the test subject APIs are described, their results are presented and discussed.
- Chapter 5 concludes the work, by exploring current limitations of the work developed, assessing the objectives attained in the proposal of this thesis, answering the research questions and stating future work.

# 2 State of the Art

This chapter consists of a state of the art on Automated Software Testing (AST), where the theoretical background of Software testing, automated testing methods and the latest academic developments are described. Additionally, Reinforcement Leaning is explored from a theoretical point of view and methods for solving search, and combinatorial problems are also described. The first subsection focuses on the methodology that guided this chapter.

## 2.1 Research Method

To help understand and answer the previously defined objectives (Section 1.3), a comprehensive systematic review was planned and managed based on the Preferred Reporting Items for Systematic Reviews and Meta-Analyses (PRISMA) methodology [13]. PRISMA is a standard aimed at helping authors achieve transparent, replicable, and complete research by defining an evidence-based checklist with a minimum set of items.

The search was conducted primarily using Association for Computing Machinery Digital Library (ACM DL) [14] as the primary source of knowledge. ACM DL is a comprehensive database of articles and literature associated with computing and information technology. The repository includes the complete collection of ACM publications, and a comprehensive database of core works in computing from other important publishers in the area, such as the Institute of Electrical and Electronics Engineers (IEEE) Xplore [15]. Regardless, since it doesn't have all documents from IEEE Xplore, this database was also queried. Moreover, the Multidisciplinary Digital Publishing Institute (MDPI) [16] database and Science Direct Database [17] were also used due to their relevance and impact on the areas related to our topic. MDPI is the largest open-access publisher database, containing more than 390 peer-reviewed scientific journals. Science Direct is a large and reputable database of scientific publications provided by Elsevier, an internationally recognised academic publishing company, that includes millions of journals, papers, and books that can be used in engineering, science, technology, medicine, and others.

In order to perform the search in the aforementioned sources, search terms had to be defined to produce a search query. The search terms were chosen after a careful analysis of the literature on this subject. To ensure comprehensive coverage of papers related to the established research questions, the employed search strategy considered Software testing, more specifically REST API testing, via reinforcement learning or search-based methods for test case generation. In addition to the fields, the most relevant keywords that the search had to include were "input generation" and "test generation". Figure 2 shows the final search query that was defined.

```
(
    ("software testing" OR "REST API Testing" OR "RESTful API testing")
    AND
    ("reinforcement learning" OR "search-based")
    AND
   ("input generation" OR "test generation")
)
```

Figure 2 – Search Query

The adopted inclusion and exclusion criteria are detailed in Table 1. Given that the theme of this study is an active research field, the inclusion criteria was limited to comprise only the most recent peer-reviewed works (2019 onwards) that focus on the presentation of test case generation applied to the Software testing field resorting to reinforcement learning or search-based methods. The search included both conference papers and journal articles but excluded survey and review papers, duplicated works, and papers that did not have the full text available.

Table 1 – Inclusion and Exclusion Criteria

| **Inclusion Criteria (IC)** | **Exclusion Criteria (EC)** |
|---|---|
| **IC1** – Peer-reviewed journal article or conference paper.<br>**IC2** – Published from 2019 onwards.<br>**IC3** – Available in English language.<br>**IC4** – Introduces a test/input generation method that is applied to the Software testing field. | **EC1** – Survey or Review papers<br>**EC2 –** Duplicate publications<br>**EC3** – Full text not available |

The search process was conducted in November 2022. To ensure comprehensive and complete coverage of papers related to the theme, our search procedure was based on two fundamental steps, the systematic search of identified databases and then the snowballing step. The systematic search was conducted by inserting the search query into the search engine of the four databases. In addition to the search query, the "the time of publication" criterion was also defined in the search engine. At this step, a total of 319 papers were identified, where the majority, 206 papers, were retrieved from ACM DL, 44 papers were found at Science Direct Database, 38 articles were discovered at MDPI database and 34 were extracted from IEEE

Xplore. Next, 1 duplicated document was eliminated, and the total of identified papers was updated to 318. These articles were then screened, which consisted of reading both the title and abstract to remove obviously irrelevant material for this comprehensive review theme. At this stage, a total of 299 documents were eliminated based on relevancy. The 19 articles left were then available for full-text analysis. The full-text analysis enables filtering for the eligibility of articles, excluding the ones that fail to conform to the inclusion criteria or that fit into the exclusion criteria. At this phase, 11 were eliminated for failing the eligibility criteria. The snowballing step began by searching the references of the 8 papers left. This check allowed the gathering of more related papers that were not retrieved initially from databases. Additionally, 3 papers were identified through snowballing. In the end, a total of 11 papers were selected for this survey. Figure 3 details the process of the search procedure according to the PRISMA methodology.

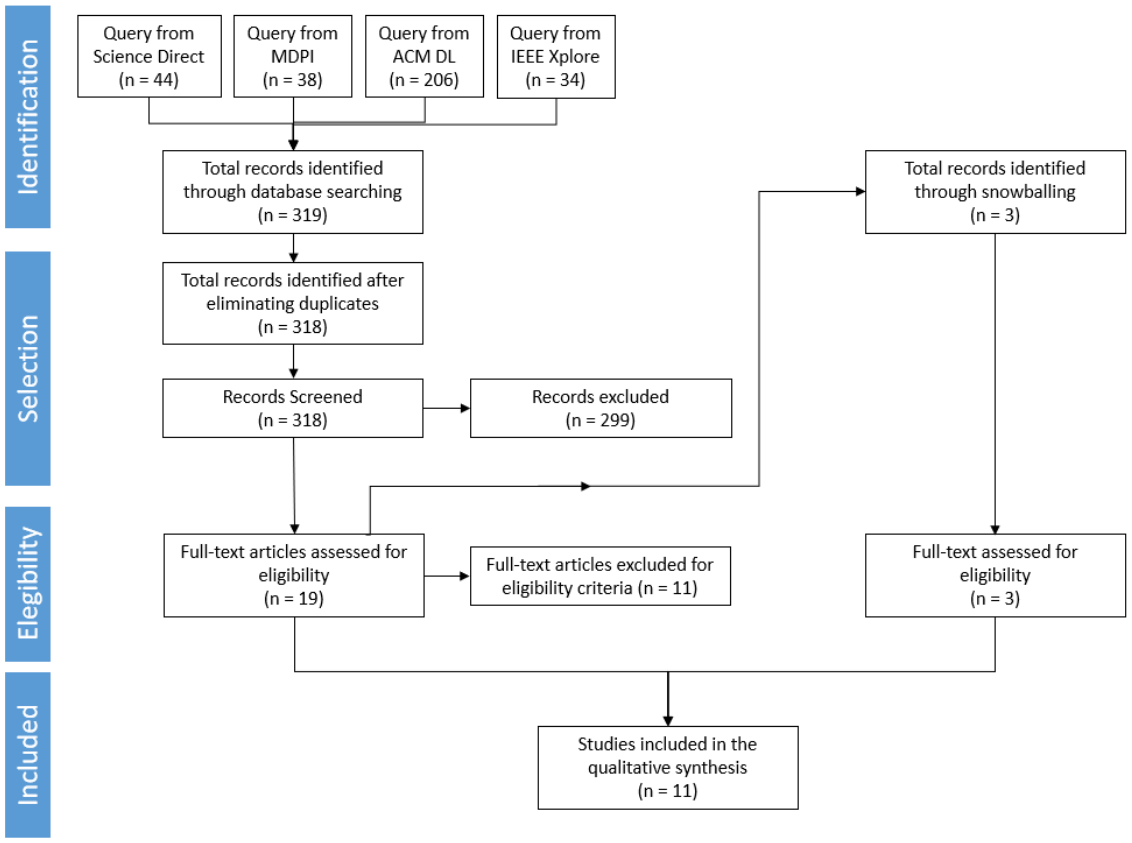

Identification
Selection
Elegibility
Included
Query from Science Direct (n = 44)
Query from MDPI (n = 38)
Query from ACM DL (n = 206)
Query from IEEE Xplore (n = 34)
Total records identified through database searching (n = 319)
Total records identified through snowballing (n = 3)
Total records identified after eliminating duplicates (n = 318)
Records Screened (n = 318)
Records excluded (n = 299)
Full-text articles assessed for eligibility (n = 19)
Full-text articles excluded for eligibility criteria (n = 11)
Full-text assessed for eligibility (n = 3)
Studies included in the qualitative synthesis (n = 11)

Figure 3 - PRISMA Diagram

## 2.2 Automated Software Testing

As defined by IEEE in [18] Software testing is the activity in which a system or component is executed under specified conditions, the results are observed or recorded, and an evaluation is made of some aspect of the system or component. Essentially, it is the process of verification and validation (V & V) of Software. It has an important role when it comes to checking whether a given system and implementation meets the user requirements and fulfils the Software quality metrics [19]. One common way to distinguish between validation and verification is by asking two questions [2]:

*Verification: are we building the product right?*

*Validation: are we building the right product?*

For these questions to be helpful, one must first understand how they can be answered. Verification can be answered by ensuring, to a certain extent, that the Software built and tested following a Software testing methodology has been correctly developed. Validation can only be answered by analysing the user requirements and satisfaction regarding the developed solution. As such, only the product owner and end-users are capable of evaluating if indeed the Software being developed satisfy all requirements and is being fulfils its requirements [1]. The answer to these questions does not ensure that the Software is defect-free or will behave as specified in every circumstance. As Edsger Djikstra famously put it [20]:

*"Testing can only show the presence of errors, not their absence."*

Software testing has a vital role in the Software development process. In many Software development methodologies, such as Test-Driven Development [21] and Plan-Driven Development [22], it appears consistently as the phase that evaluates the developed work and reveals defects of a system. It evaluates correctness, completeness, and accuracy of the Software. Testability is the scope of the ease with which a system can be tested. It is a very well-established Software quality attribute that highly influences the quality of the Software. However, it being very much overlooked and along with the lack of testing, these end up being the main reasons for the failure of big projects [19]. This usually happens because Software testing is a process that takes a lot of time, and the discovery of errors in previous functionalities can lead to a delay in the development of new functionalities. Nevertheless, the raise of errors and defects that appear via Software testing should not be disregarded nor considered an adversity towards the Software being developed. As stated in [19]:

*"Testing is not a pit, it is a ladder!"*

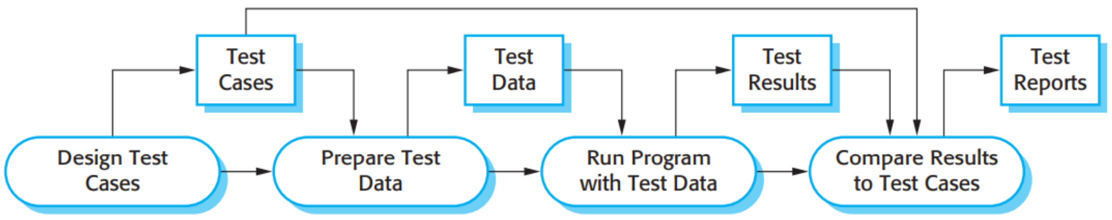


Figure 4 – A model of the Software testing process [2]

In an attempt to reduce the time-consuming task that is testing Software, automated Software testing tools have been developed and are nowadays highly used along with the manual testing process. Figure 4 depicts the typical Software testing process that a system must go through.

While manual testing fits perfectly in this model, automated Software testing lacks the ability to automatically generate traditional test cases, since human expertise of the system must be involved to specify the expected output [2]. Therefore, an AST should comprise a different oracle that does not require human intervention. In addition to reducing the testing time, they are also characterised by increased fault detection, human-effort alleviation and increased confidence [23].

AST is often divided in three types: (i) Black-box, where the tool only has access to input specifications and (ii) White-box, leveraging from knowing the source code [24] and (iii) Grey-box, benefitting from using both Black-box and White-box techniques [25]. The former is a method of evaluating the functionality of a Software system without having access to the system's internal workings, nor knowledge of it. This type of testing is based on the end user's requirements and is conducted from the customer's perspective. By providing both valid and invalid input, black box testing can identify any incomplete or unpredictable requirements and address them [26], [27]. White box testing is a method of testing that involves examining the internal structure and logic of a Software program. Therefore, it is usually carried out by the developers. It is commonly used to detect logical errors in the code, debug the code, and find typographical errors. This approach to testing is also helpful in uncovering incorrect programming assumptions and verifying that the code follows coding standards and best practices [27], [28]. Grey-box testing is the combination of the former types described. It leverages from combining black-box and white-box techniques that the tester can use to design more effective and efficient test cases [25].

Regardless of the type, the test cases differ from one another depending on the generated input and the action sequence taken, since different combinations can lead to different path exploration of the SUT. The input can be generated by resorting to multiple methods, which will be further discussed in a systematic manner of input/test generation subsection of this chapter. This thesis only accounts the Black-box testing type for further discussion in the context of APIs.

Software testing is comprised of a set principles and levels, which provide guidance and method to the developed test cases. AST tools should also consider these principles and levels to provide different valid test cases which are automatically generated. The following subsection explores the principles and levels of Software testing.

### 2.2.1 Principles and Levels of Testing

Software testing principles are the fundamental concepts that form the foundation of effective Software testing. By following these pillars, Software testers are guided to perform test cases, which ultimately will evaluate the quality and functionality of a Software application or system. The seven principles of Software testing described in [19] include:

1. Testing shows presence of Defects – Testing highlights the defects; it does not present the absence of them. As such, the goal of testing is to make the Software fail.

2. Exhaustive testing is impossible – Testing each input for every possible value is impractical and unfeasible. Testing should be made considering the domain boundaries of an input.
3. Early Testing – Defects should be found in the early phases of the Software Development Cycle, as it will reduce the cost of amendment.
4. Defect Clustering – Small modules are more likely to have the highest number of defects.
5. Pesticide Paradox – Running the same test cases multiple times will most likely not produce any different outcome. Therefore, these must be regularly revised, adding new test scenarios.
6. Testing is context-dependent – Different Software requires different types of testing.
7. Absence of Errors-Fallacy – Bug-free Software can still be unusable if the wrong requirements were incorporated in the Software and still do not address the business needs.

In addition to the importance of following the principles of testing, a system can also be tested on multiple levels, from more fine-grained testing where only units are under test to more general testing simulating the actions of a user. Additionally, the end user's interaction with the system is also considered validation testing of the system. From a simpler and fine-grained to a more complex and general testing method, the levels of testing are as follows [19]:

1. Unit Testing – The system is broken down into its smallest testable parts, designated units. It consists of independently testing each of those units.
2. Integration Testing – The purpose of integration testing is to expose defects in the integration between multiple units.
3. System Testing – Focusing on the entire Software, this is usually the last test performed by developers before making a system public. This level contains various types, such as: (i) usability testing; (ii) performance testing; (iii) security testing and (iv) Regression testing.
4. Acceptance Testing – The last level of testing, which is performed by the end-users, reflects the acceptability of the system. This level regards different types of tests, such as: (i) Alpha testing; (ii) Beta testing; (iii) user acceptance testing; (iv) business acceptance testing; and (v) exploratory testing.

Testers and automated Software testing tools should consider these principles and levels of testing to provide the best possible outputs to whom they may concern. The lack of testing at any of these levels can jeopardise the quality of the Software being developed [29].

Most of the test levels mentioned, are typically made in a white-box manner. However, because of this there is a certain bias in the tests to assess their validity with correct input and analyzing the expected output. However, faulty input should also be considered, because it can raise unknown vulnerabilities of the system, which if not mitigated can be exploited by potential malicious users in order to disrupt the application or service. Creating manual testing for each possible faulty input is extremely difficult and not feasible, since an apparently normal input can be considered faulty to the system for numerous reasons. As such, analysing this problem from a black-box perspective, even though it raises other problems, such as the search and combinatorial one presented in the first section, can also be beneficial, since it allows creating and executing multiple test cases of mutated faulty input.

Following the scope of this thesis, the following subsection describes and analysis black-box testing methods, along with the latest academic developments towards automation of this testing type.

### 2.2.2 Black-box Testing

Black-box testing is a type of testing conducted from the end-users perspective and tries to identify any ambiguities or inconsistencies in the requirements specifications. It does not require testers to have in-depth knowledge of programming languages or implementation details, allowing for greater independence between programmers and testers. Overall, the main advantage of black box testing is its ability to evaluate the functionality of a Software system without requiring knowledge of its internal workings. However, because of this lack of knowledge, efficient automation can be extremely challenging, as there must be mechanisms for generating the right input to cover most of the test cases of the SUT, to test its resiliency and correctness.

Statistical test methods, such as random testing, can provide valuable information about potential failures or defects in a system. However, this information is probabilistic in nature and can be difficult to interpret in terms of the reliability of the test. Additionally, random testing may not be effective at detecting rare failures. Despite these limitations, statistical testing is a commonly used approach, particularly because it generates large amounts of quantitative data. It is also often efficient in terms of time and resources compared to other testing methods, due to the simplicity of generating random input data. While it has its limitations, it is important to continue exploring ways to improve the quality and reliability of random testing and address the issue of evaluating probability and reliability [30].

However, one should first understand the existing Black-box testing methods in order to comprehend how these have been and can be utilised by other authors to achieve black-box testing automation. As such, section 2.2.2.1 overviews and compares the already existing black-box testing methods, whilst section 2.2.2.2 consists of a systematic review of the automation of black-box Software Testing.

#### 2.2.2.1 Testing Methods

Test cases are created using different testing techniques in order to be more thorough and effective. This helps ensure that the Software is complete and that the test conditions chosen have the highest likelihood of uncovering any errors. Rather than randomly selecting test cases, testers can use these techniques to design their tests in a more organised and structured way. Additionally, combining various testing techniques often yields better results than relying on just one technique [31]. Software testing techniques are divided into 5 groups: (i) Intuition and Experience based, (ii) Specification-based, (iii) Code-based, (iv) Fault-based, and (v) Usage-based [32], [33]. Naturally, not all of these techniques can be automated in a black-box testing environment. For instance, code-based testing is typically a white-box technique that requires code analysis and intuition and experience-based specifically requires human expertise. Nonetheless the remaining techniques are compatible with the described context. Therefore, considering the scope of this thesis, testing methods of the remaining techniques will be further described based on several articles [1], [2], [34].

#### 2.2.2.1.1 Equivalence Partitioning and Boundary-value Analysis

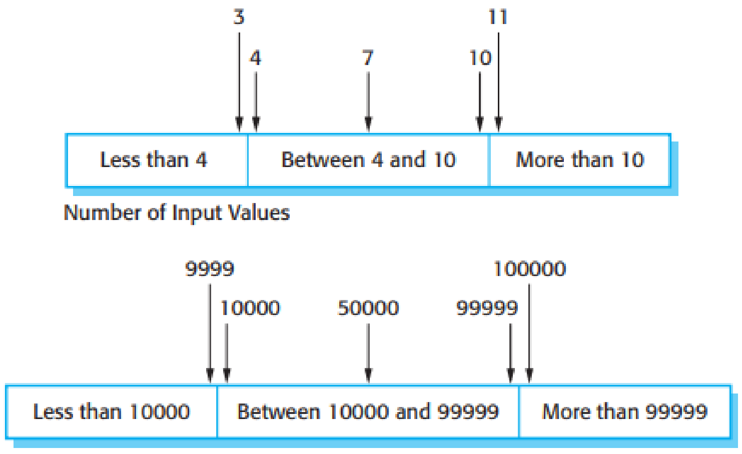


Figure 5 – Equivalence Partitioning and Boundary-value Examples [2]

Input data can be divided into distinct partitions, and Software often behaves the same way for all members of a particular partition. By dividing the input data into diverse categories based on the domain specificities, as shown in Figure 5, a developer can test a large portion of it without having to test the entire system. One challenge with this approach is finding an effective way to automate the partitioning of the input domain. However, partition testing is a promising option for producing good results and ensuring a high level of robustness in the implementation. After establishing the partitions, only the actual boundary value and the values directly below and above it has to be tested in order to understand the behaviour of the system.

#### 2.2.2.1.2 Cause-Effect Graphing

Cause-effect graphing involves populating a graph that represents the relationship between causes (input data or conditions) and effects (output or behaviour of the SUT). This technique is used to select test cases that cover a large portion of the input domain and to identify the most relevant combinations of input data to test. It can also help testers understand the behaviour of the SUT and identify potential failure points.

#### 2.2.2.1.3 Error Guessing

Error guessing is a testing technique in which the tester uses their own expertise in the field, knowledge, and intuition to predict where errors or defects might occur in the Software being tested. This technique involves identifying areas of the Software that are likely to have problems, based on the tester's understanding of the specifications of the SUT. It is an informal and subjective testing technique that relies on the tester's expertise and judgment. As such, it can be extremely hard to be automated [1].

#### 2.2.2.1.4 Random Testing

The name of the technique is self-explanatory. Input values are picked at random, originating multiple test cases. Although this technique may not always be the best technique to ensure thorough testing of a system, as it may never explore some hard-to-reach functionalities, it covers wide input ranges, which permits establishing the limit of failure frequency, even if not necessarily finding all faults. Hence, Random Testing can potentially be used in automated Software testing.

#### 2.2.2.1.5 Exhaustive Testing

This method involves testing every possible combination of inputs. The combination of all possible inputs in a black-box manner is not feasible since the range of inputs may be never-ending. However, one should still consider this technique if the domain is small (e.g., Booleans).

#### 2.2.2.1.6 Use Case Testing

This technique involves understanding the specifications of a system, regarding the functionalities it possesses and testing multiple sequences of functionalities. A change of state in the SUT by performing certain operations may disrupt other functionalities. This technique ca be integrated in an automated context. However, in practice, depending on the size of the SUT it can lead to high computational cost.

### 2.2.2.2 Literature of Automated Black-box Software Testing

Nowadays, web APIs are the de-facto standard for Software integration [35], being most Software made available via RESTful APIs. As RESTful APIs gain momentum, so does their testing. Recent advances in the literature show that black-box testing of RESTful APIs has outputted effective results and is capable of contributing to the validity, reliability and correctness of these systems.

In this section the findings gathered during the research outlined in Section 2.1 are described in greater detail. Table 2 summarises the target API information, best input search method, if any, input generators and the type of test case generators that the authors found most useful to perform automated black-box testing.

Table 2 – Methods for black-box Software testing

| Articles | Target API Information | Input Search Method | Input Generators | Test Case Generator |
|---|---|---|---|---|
| [36] | OpenAPI Specifications | - | Custom Data Generators + Public Knowledgebases + Output Data | Fuzzy Testing + Adaptive Random Testing + Constraint-based testing |
| [37] | OpenAPI Specifications | - | Random Data Generators + Custom Data | - |
| [38] | OpenAPI Specifications | Breadth-first search + Random Walk | Use Case combination + Random Input | Fuzzy testing |

| | | | | |
|---|---|---|---|---|
| | | | Generator + Output Data | |
| [39] | OpenAPI Specifications | - | Random Input Generator + Random Input Generator | Specification-based testing |
| [40] | OpenAPI Specifications | - | Random Input Generator | Property-based testing |
| [41] | - | Similar to Genetic Algorithm (GA) | Random mutation of faulty real user input | - |
| [42] | OpenAPI Specifications + RESTful-service Property Graph | - | Random Input Generation + OpenAPI Specification input example + Data output | Model-based testing |
| [43] | Navigational model | Input Distance Formula | Random Input Generation | Diversity-based testing |
| [44] | OpenAPI specifications | Not specified but the "bots" learn patterns | Data Ouput + JSON Perturbations + Random Input Generation | Model-based specification-driven testing + Metamorphic testing + AI-driven testing |
| [45] | - | RL + Monte Carlo Control (MCC) Algorithm | Random input Generation + Guided input generation | Property-based testing |
| [46] | Navigational Model | RL + Deterministic Finite Automaton (DFA) | - | Curiosity-driven testing |

Regarding data generation from a black-box testing perspective, since there is neither knowledge nor access to the internal working of the regarded SUT, authors have found ways of constraining the input generation and testing action sequences descriptive information regarding the API under test, which is publicly available. For instance, in [36], the authors decided to test a REST API. The authors decided to produce constrained input via three different methods: (i) OpenAPI specifications [47] (formerly known as swagger), which describes the methods for interacting with a certain API; (ii) custom data generators which produce constrained input; and (iii) public knowledge bases. Similarly, [37] considers the OpenAPI specifications to interact with the system and additionally can generate automatically test cases, but also allow the user to influence the generation process to introduce human expertise and specific context. Additionally, the authors use functional and non-functional metrics to analyse the performance of the APIs.

RESTler [38] is also an approach to REST API testing, which focuses on fuzzy testing to test the security of the system. The tool analyses the API specifications and generates a sequence of requests by inferring order dependencies and analysing dynamic feedback of prior tests. In conclusion, the tool was capable of detecting 28 bugs in various cloud services, which were confirmed and tested by service owners. Ed-douibi et al. in [39] also present an automated REST API testing tool that relies on OpenAPI specifications to generate specification-based test cases to ensure APIs meet the requirements defined in their specifications. Their tool was validated with 91 OpenAPI definitions, and the experiments showed that the generated test cases cover, on average, 76.5% of the elements included in the definitions, with 40% of the tested APIs failing. Karlsson et al. in [40] analyse the behaviour of a RESTful API by using automatic property-based tests generated from OpenAPI documentation. In their work, input is generated randomly. Similarly, in [41], the authors propose using the data collected from a public API and mutate faulty input to produce valuable test cases. Although these approaches can diminish the input space, producing realistic test data, and discover web APIs vulnerabilities, they do not keep track of the path exploration of the SUT, which might can be a major drawback of these approaches, as they might not sufficiently cover the entirety of the SUT test paths.

The authors of [42] present Morest, their model-based RESTful API testing technique that uses a dynamically generated and self-updating RESTful-service Property Graph to model the behaviour of RESTful services and guide the call sequence generation. Their empirical evaluation led them to conclude that Morest was capable of successfully requesting a higher average of API operation, cover more lines of code and detect more bugs than state-of-the-art techniques. In [43], the authors produce a navigational model of the web application by crawling it to discover possible test cases. The presented method's first test is generated randomly and is added to a set of executed tests. The subsequent tests and concrete input vectors are selected depending on the distance between each candidate and the current set of executed test cases, where only the farthest case is computed. Their goal is to generate a set of test cases that diversify the coverage of the navigational graph and the use of input data. Although they present a method that does not rely on exhaustive testing of the API that requires in-browser executions, they still rely on random input which dictates how long the system will take to find the next valuable input data to cover different test paths, since it is calculated by the distance formula. Lastly, in [44], the authors also resort to three different methods for testing web APIs: (i) resorting to search-based methods; (ii) mutating API JavaScript Object Notation (JSON) input and output; and (iii) computing metamorphic testing. These methods naturally preserve the quality of the initial data since they don't tend to diverge from the initial input.

However, more intelligent solutions have also arisen. Specifically, the ones using RL to solve the search-based problem that is finding the right input and combination of use cases. The authors in [45] focus on improving valid test input generation relying on RL to perform such a task. They motivate their work by stating that Property-based testing is a great way of quickly generating a huge amount of test cases and running them through a parameterised test driver. However, when the test driver (e.g., web API) requires validity constraints, random input generation fails to generate enough valid inputs. Although they could rely on white-box or grey-box information to solve the problem, this would decrease the speed at which tests could be executed. As such, they present a tabular guide, which limits the input space within a random generator. Their guide is based on the MCC learner, which defines the policy and the choice for a given state, which will allow the generator to generate the input value. In conclusion, their system does not scale so well because of the complexity of the MCC algorithm and because of the RL method chosen, which was the Q-table. Additionally, RLChecker, their tool, had difficulty generating the first valid input value for very strict domains. In comparison to other tools, RLCheck obtained a better average time to discover and reliability percentage to bug finding. However, it was not capable of finding all bugs. The authors in [46] present WebExplor, which is an automatic end-to-end web testing framework which adopts curiosity-driven reinforcement learning to generate action sequences, following a Use Case testing method, since many times a sequence of functionalities of a system can raise unexpected defects. This framework leverages RL to perform adaptive exploration of web applications and generate action sequences. Additionally, the authors tackle local optima by introducing a Deterministic Finite Automaton (DFA) which continuously records the transitions and states visited during the RL exploration. This way, when RL gets stuck, WebExplor selects a path from the DFA based on the curiosity. In conclusion, their framework uncovered 3466 exceptions and errors in a real-world commercial web application.

Regardless of their usefulness, none of the solutions presented are comparable to the framework being developed in this work. None of the described tools combines RL for input and action sequence generation and API specifications along with user input to create more diverse test cases and perform more thorough testing of RESTful APIs to discover hard-to-find combinations to search for code defects. Moreover, none of the solutions found utilize fuzzy testing mutation methods, such as the ones that fuzzers usually integrate, in order to try to find vulnerabilities in the APIs.

### 2.2.3 Fuzzy Testing

Fuzzy testing is a vulnerability discovery technique which is typically used for automated Software vulnerability mining [48]. Recent applications show that this technique has shown great results for uncovering Software defects [49]–[51]. The premise of this technique is to generate malformed input that is used to execute use cases and monitor the SUT for possible errors or bugs. Figure 6 describes the process of fuzzy testing.

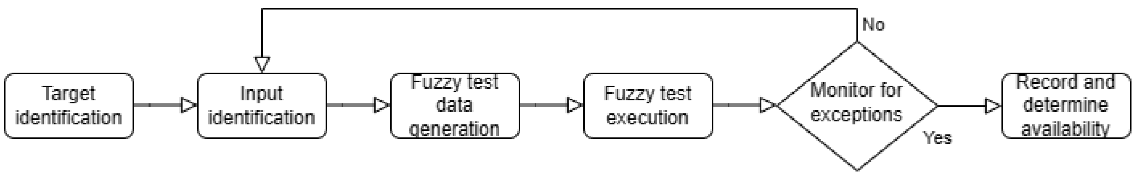


Figure 6 – Fuzzy Testing Flowchart

The first step of fuzzy testing, as depicted in Figure 6, is the target identification, where the fuzzer must know the specification of the endpoint to interact with it. This way, the fuzzer moves onto the second phase, where the input combination and typing are now known. The fuzzer then starts generating the malformed input, usually resorting to random or mutated input, and executes fuzzy tests on the target. The fuzzer then proceeds to repeat this process on the known target until exceptions are found. After finding the exceptions, these are recorded so that security personnel can assess and report the uncovered vulnerabilities [48].

Fuzzy testing's ease of deployment and lightwheightness are often considered advantages of this testing technique. Moreover, tests are usually executed quickly and efficiently, with a great adaptability to modern large-scale Software [48]. However, this technique often shows low code coverage due to its randomness, along with poor execution efficiency due to excessive redundant tests.

Fuzzing Software systems can be made with multiple objectives, including test the robustness, performance, correctness and fault tolerance of a SUT. Therefore, fuzz testing is divided into 4 main methods, each comprised of their own characteristics and goals: (i) Mutation testing, (ii) Fault injection testing, (iii) Robustness testing and (iv) Stress testing. The first involves introducing small, controlled changes to the code being tested and then evaluating the effectiveness of the test suite in detecting those changes. It is great for assessing the quality of a test suite by measuring its ability to detect code changes. Fault Injection involves introducing malformed input into the code being tested in order to evaluate the SUT ability to handle those faults. Robustness testing is one that evaluates the SUT's ability to continue functioning properly in the presence of hardware of Software failures, exposing SUT to multiple different conditions. Lastly, Stress testing focuses on identifying performance bottlenecks and potential failure points in a system.

Recently, fuzzy testing has gained some attention for its ability to uncover vulnerabilities and code defects which typically are not found in white-box testing through manual testing. Therefore, multiple frameworks have emerged to assists developers and testers to uncover Software vulnerabilities, which in turn lead to the creation of multiple fuzzers of different types. Fuzzers can be considered naïve or non-naïve, with the former being fairly easy to implement, however, because of its blindness towards the SUT they are unlikely to achieve interesting results in a timely manner. Modern fuzzers improve on naïve fuzzers by introducing some sort of information about the target system, making them grey-box fuzzers. According to [52], these can be of three types: (i) Mutation-based, (ii) Generation-based, and (iii) Evolutionary.

The following subsections describe the different types of non-naïve fuzzers along with a description of the existing frameworks.

#### 2.2.3.1 Mutation-based fuzzers

Mutation-based fuzzers are characterized by their capability to perform mutations on test input. Typically, fuzzers of this type are not aware of the input format or specifications and therefore cannot perform wise mutations on the data. Peach [53] is an example of a fuzzer of this type. It is capable of performing both smart and naïve fuzzing, and it includes a monitoring system allowing for fault detection, data collection, and automation of the fuzzing environment.

Peach is highly adaptable to the Software being tested since it adapts to any data consumer. As such, it is typically used to fuzz file formats, network protocols, and APIs. This tool works based on configuration files, usually named Peach Pit files, that describe the structure, data types and establishes relationships about the Software being fuzzed [54]. Although this fuzzer is described as a mutation-based fuzzer, it can also perform generation-based fuzzing.

#### 2.2.3.2 Generation-based fuzzers

Generation-based fuzzers resort to specifications of the SUT to gain some insight regarding the format of the input or protocol. This kind of fuzzer can generate inputs based on these specifications and therefore produce more realistic input to perform better tests and achieve interesting results faster. In addition to Peach, BooFuzz [55] is also a fuzzer of this type, considered a network protocol fuzzer. Network protocol fuzzing focuses on sending erroneous, unexpected, or malicious protocol data to a system via the network in order to find weaknesses and potential security problems in the protocol.

Boofuzz is the successor of Sully [56] fuzzing framework. Besides numerous bug fixes, it also expands on the original version of the fuzzing framework, enhancing performance, ease-of-use, data instrumentation and making highly extensible. This fuzzer's goal is to simplify not only data representation but also transmission and instrumentation. This framework is capable of watching and maintaining records regarding network information, whilst providing target monitor and response methods. Moreover, considering time complexity, it scales horizontally to allow better fuzzing [56]. This way, BooFuzz requires little to no interaction, allowing vulnerability researchers to focus on other areas of exploitation [55].

#### 2.2.3.3 Evolutionary-based fuzzers

Evolutionary-based fuzzers are the most recent type of fuzzers to appear. These build on mutation-based fuzzers by selecting inputs over others for mutation. These typically resort to an evolutionary algorithm to evaluate the outcome of a certain input and proceed based on that evaluation. State-of-the-art fuzzers of this type typically define a fitness function to select the best-ranked inputs to mutate. These include Honggfuzz [57], American Fuzzy Lop (AFL) [58], and libFuzzer [59].

Honggfuzz is a security oriented, feedback-driven fuzzer based on code coverage, with a complete set of analysis tools. Similar to BooFuzz, it too works in a multi-threaded way to avoid overhead and instead scale horizontally. It also keeps track of records and provides visual interface for analysis. Given an initial input corpus, it identifies which are valuable for code coverage increase and stores them in an in-memory database. Then it performs random mutations on that population and begins a new fuzzing round, in which the newly mutated data is utilized for fuzzy testing [60].

AFL is an open-source mutational coverage-guided fuzzing framework that was designed to automate the process of generating and injecting random or semi-random data into a SUT, with the goal of testing robustness and resilience by uncovering potential buggy program points. AFL uses a mutation fuzzing technique along with an evolutionary algorithm to generate test cases, which consists of taking a set of initial inputs and applying several mutations to generate a large number of new test cases. These test cases are then fed into the system or Software being tested, and the results are evaluated to identify any potential issues or vulnerabilities [51].

libFuzzer is also an example of a coverage-guided, evolutionary-based grey-box fuzzer that generates mutations on the initial data in order to maximize code coverage. This fuzzer also adapts quite well to the SUT, since it does not require knowledge regarding the Software [60]. Similarly to AFL, this fuzzer also keeps track of the mutated inputs that increase code coverage for later use and uses multiple threads to maintain fast performance. For random input generation, it combines bitwise, mathematical, delete and overwrite operations [61].

## 2.3 Artificial Intelligence

Artificial Intelligence (AI) is the field of computer science focused on creating intelligent machines that can achieve specific goals. According to [62], intelligence can be defined as the ability to accomplish tasks effectively through computation. This definition applies to a range of beings, including humans, animals, and machines.

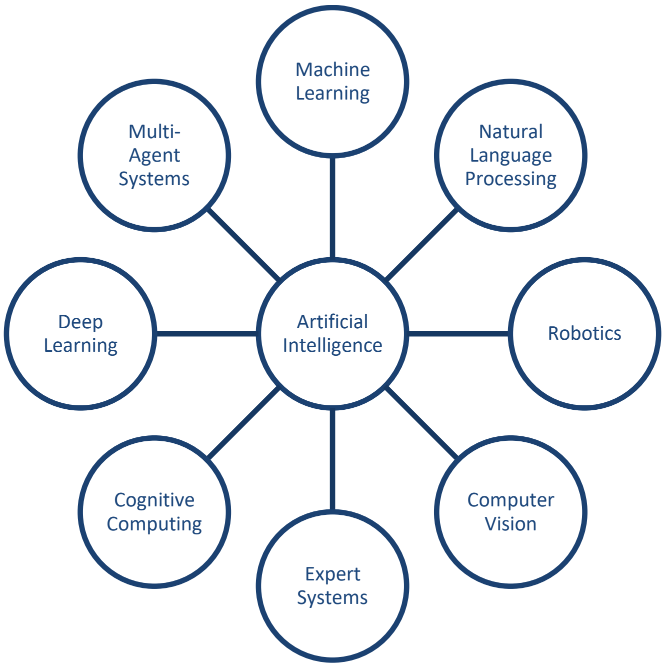


Figure 7 – Artificial Intelligence Fields

AI appears in many shapes, ranging from the most straightforward formats to more complex mathematically computerised concepts, such as rules and neural networks. AI has recently been heavily adopted by the industry in many fields and continues to gain attention, making this technology more relevant. As shown in Figure 7, within AI, there are multiple fields [63], each with its own purposes. Depending on the problem at hand, each field has a set of algorithms and methods that help problem-solving. The complexity of the algorithms used is directly dependent on the complexity of the problem to be solved, and sometimes, the solution is not finding all possible cases in highly complex problems but instead trying to find the best possible [64]. Moreover, one has to consider the environment in which the AI is applied. Volatile domains usually vary a lot. As such, the application of AI in these must be performed alongside the changes. This means that an algorithm created for a certain condition in the domain is no longer valid if the domain changes [64]. In these cases, the solution may be using learning algorithms, which are referred to as Machine Learning (ML).

ML has the ability to interpret data to solve previously impossible tasks. ML algorithms allow a system to gain the ability to learn without being confided to explicitly defined rules. As described in [65]:

*"Programming computers to learn from experience should eventually eliminate the need for much of this detailed programming effort."*

ML algorithms can be defined in 3 different subcategories, each with distinct approaches suited for distinct types of problems, which are: (i) Supervised learning, (ii) Unsupervised learning and (iii) RL. The first incurs in a training phase where labelled data is fed into the model so it can learn to classify unseen data properly [66]. The second does not require any training data. Instead, it observes and segments the data by the number of classes specified [67]. Recently these two have also been combined to produce semi-supervised learning. This method takes a certain amount of data already labelled and attempts to use unsupervised learning to label the rest of the data. By comparing the classifications with the already correctly labelled data, the algorithm can adjust its classifications accordingly [68]. Lastly, RL enables AI-based agents to learn complex and dynamic environment by trial-and-error, aiming to find optimal actions to be taken in certain events [69].

In this thesis, RL is utilised to optimise input generation and action sequences of RESTful APIs functionalities in order to perform more thorough testing of these systems and discover the hard-to-find paths to look for code defects. As such, the next sections consist of a theoretical overview of RL and a review of algorithms that are indicated to solve search and combinatorial problems.

### 2.3.1 Reinforcement Learning

RL is a subfield of ML that deals with decision-making and problem-solving in environments where an agent learns by interacting with its environment and receiving feedback in the form of rewards or punishments. One of the key features of reinforcement learning is that in an RL environment, an agent is capable of learning through trial and error without the need for explicit supervision or labelled training data. This makes it particularly well-suited for solving problems in underlying volatile domains.

In recent years, RL has had much success in multiple applications, the most relevant being control, recommendations systems and games [70], [71]. However, their configuration is not as straightforward as one might think, as there are still challenges that one must tackle to implement an RL environment. One of the key challenges in RL is the exploration-exploitation trade-off, which refers to the tension between exploring new actions and states in order to learn more about the environment and exploiting known good actions in order to maximise reward [70]. Another challenge is the sample efficiency of learning algorithms, which refers to how quickly they can learn from a given amount of experience. In order to scale up reinforcement learning to real-world applications, it is important to develop algorithms that are both sample efficient and able to balance exploration and exploitation effectively [70].

In order to understand RL, one must first understand Markov Decision Processes (MDP) since RL functions are based upon that [70].

#### 2.3.1.1 Markov Decision Processes

A Markov decision process (MDP) is a mathematical framework for modelling decision-making situations which are used as world representations where agents can learn to interact with it to find optimal solutions to deal with stochastic problems. It consists of a set of states, actions, and a transition function that defines the probabilistic transitions between states based on the actions taken. As described by [72], it is a *"non-deterministic search problem"*.

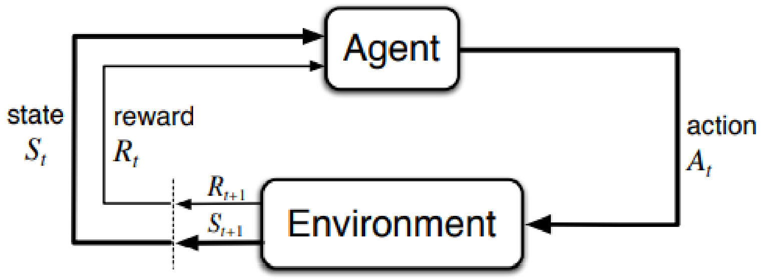


Figure 8 – RL Algorithms Flowchart [70]

As depicted in Figure 8, at each time step, the agent observes the current state of the environment and selects an action based on its policy. The environment then transitions to a new state based on the action taken, and the agent receives a reward based on the new state. The goal of the agent is to learn a policy that maximises the expected cumulative reward over time.

MDPs are often formally defined using the following components [73]:

- $S$ – Finite number of states;
- $s_0$ – Initial state;
- $A$ – Finite number of actions;
- $T(s,a,s')$ – Possible transitions from a state s to the next, $s'$, carrying out an action $a$;
- $\delta: S \times A \rightarrow \text{dist}(S)$ – Function that given a state $s$ and an action $a$, possible to take in state $s$, returns the probabilistic distribution of which possible states the transition can be made to and the associated probability for each;
- $R(t)$ – Reward or loss associated with a transaction $t$.

Additionally, its concepts can be defined as such [73]:

- **Policy: π*: *S* → *A*** – A policy is a strategy used to choose the best action to make in each moment, for each state *S* a policy $\pi$ attributes an action *A*;
- **Utility** – Some of the rewards obtained with the actions already taken;
- **Values** – Expected maximum utility for each state, assuming that all actions taken are ideal;
- **Q-Values** – Expected maximum utility for each state assuming that the first actions taken is uncertain and therefor might not be ideal, and all other following actions taken are ideal, the highest Q-value for a state must correspond with the value for that same state.

In order to solve an MDPs, the optimal policy that maximises the obtained utility at every moment must be found. There are multiple processes to do this that are based on finding the values and Q-values aforementioned. Bellman equations [74] are used to find these values. These equations may vary depending on the specific problem.

To solve MDPs, other mathematical work approaches can be utilised. RL is not exclusive to solving these problems. However, focusing on the scope of this work, only RL is regarded. Moreover, being non-stationary, these algorithms are ideal for problems that may require some trial-and-error operations to reach a certain path, for instance, in Software testing to cover most paths of a black-box RESTful API. As such, in order to select the right algorithm, one must first comprehend the taxonomy of RL algorithms to correctly review and assess algorithms to solve the challenges of this thesis. Figure 9 describes the taxonomy of RL algorithms, based on [75].

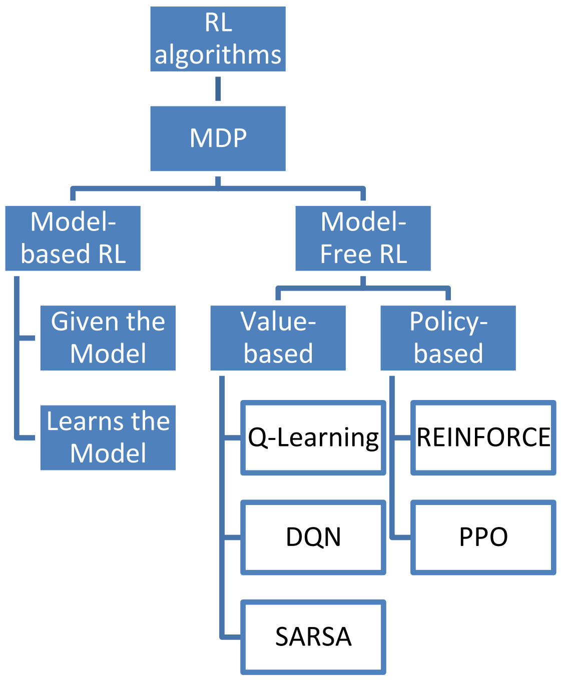


Figure 9 – Reinforcement Learning algorithms Taxonomy

MDPs can be of two different types: (i) Model-based or (ii) Model-free. Model-based MDPs consist of consulting a model that describes the environment to be able to take an informed decision, without failing so much into trial and error. This model can be built from observing the outcomes of certain actions taken in a certain environment. Model-free MDPs, on the other hand, do not require any model. In this case, the environment in not so well known, as such, no action outcome can be predicted. As per described in Figure 9, Model-free MDPs are further divided into: (i) Value-based, and (ii) Policy-based. Where in the first the policy is explicitly defined and kept in memory during learning, and in the second the policy is implicit in the value function, picking the actions with the best value [75].

In the summarized taxonomy presented in Figure 9, each algorithm behaves differently from one another. Each define different ways of processing rewards, which will produce different outcomes and require unique implementations. In order to understand each algorithm that belong to different branches of the presented taxonomy, the following subsections consists of their description.

#### 2.3.1.2 Q-Learning

Q-learning is typically used to learn the value of taking different actions in different states. It is a value-based method, which means that it estimates the value function for different states or actions and uses this value function to determine the optimal policy [76], [77].

The core idea behind Q-learning is to estimate the quality of each action, represented by the Q-value, which is the expected cumulative reward for taking that action in a particular state. The Q-values are updated iteratively based on the observed rewards and the estimated Q-values of the next states. Algorithm implementation resorts to a Q-table, which functions as a lookup table that automatically stores the Q-values for all states and actions in each transition [77].

#### 2.3.1.3 State-Action-Reward-State-Action

State-Action-Reward-State-Action (SARSA) is a model-based reinforcement learning algorithm used to learn the optimal policy for an agent in a given environment. This algorithm is based on Q-Learning. However, it uses the expected value of the next action rather than the maximum expected, to update the Q-values, as shown in the equation below [70].

$$\mathrm{Q}(S, A) = \mathrm{Q}(S, A) + \alpha[R + \gamma \mathrm{Q}(S', A') - \mathrm{Q}(S, A)]\,, \tag{1}$$

where *S* is the current state, *A* is the current action, *R* is the immediate reward, *S'* is the next state, *A'* is the next action, *y* is the discount factor and $\alpha$ is the learning rate. This Q-value serves as the measure of the expected cumulative reward that the agent will receive from a certain state-action pair, given its policy [70].

Contrary to Q-Learning, this algorithm is best suited for stochastic environments where future states and rewards may not always be the same [70].

#### 2.3.1.4 Deep Q-Network

Deep Q-Network (DQN) is an algorithm that combines Q-learning with neural networks [78] to continuously understand the environment and compute the Q-values for all possible actions in a given state [79].

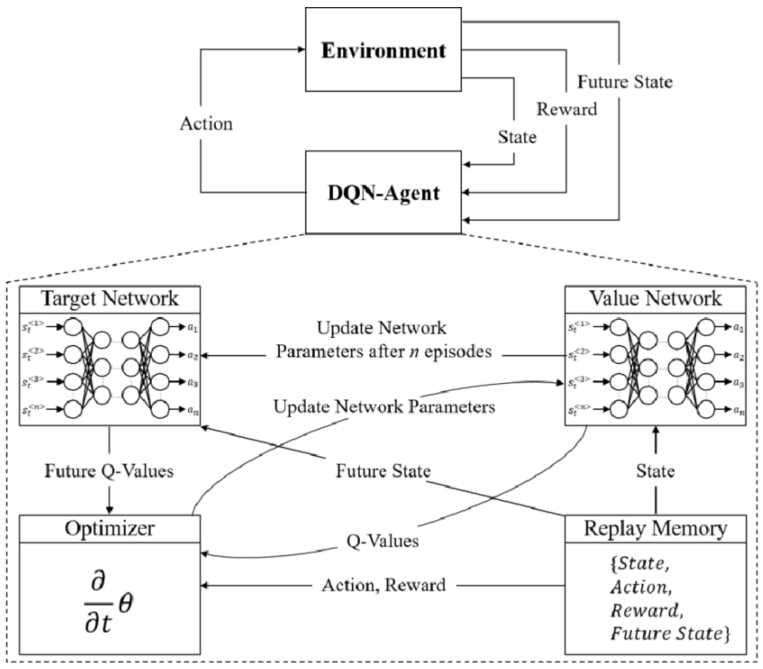


Figure 10 – Deep Q-Network Architecture [80]

As depicted in Figure 10, to approximate the Q-function, DQN uses the neural network rather than the formerly mentioned Q-table to store the Q-values. The neural network takes the state of the environment as input and produces a vector of Q-values for each possible action as output. The Q-values are then used to determine the optimal action for the agent to take in the current state. This algorithm uses techniques such as replay memory to store past experiences and fixed Q-targets which consists of using two neural networks: (i) one that is updated during the learning process, (ii) and one that is used to compute the target Q-values, which allows for a more stable learning process. The Q-values are updated according to a modified version of the Bellman equation [80], [81]:

$$\mathrm{Q}'^{(S_t,A_t)} = \mathrm{Q}(S_t,A_t) + \alpha[R_{t+1} + \gamma * max_a[\mathrm{Q}(S_{t+1},\mathrm{A})] - \mathrm{Q}(S_t,A_t)], \qquad (2)$$

where $\mathrm{Q}(S_t,A_t)$ and $\mathrm{Q}'(S_t,A_t)$ are the recent and updated Q-values for all possible actions $A$ in state $S$ with a time step of $t$, α the learning rate and $R_{t+1}$ the resulting reward. γ works as a discounting factor and $max_a[\mathrm{Q}(S_{t+1},\mathrm{A})]$ represents the prediction of the best action for the future state $S_{t+1}$.

### 2.3.1.5 REINFORCE

REINFORCE algorithm is mostly used to learn a policy directly without estimating the value function. It is a policy-based method, which means that it learns a policy that maps states to actions and does not estimate the value of taking different actions in different states [70], [82].

The core idea behind the REINFORCE algorithm is the optimisation of the policy using gradient ascent by adjusting the policy parameters in the direction that increases the expected reward. The policy parameters are updated using the observed rewards and the gradient of the expected reward with respect to the policy parameters. This algorithm can be implemented using, for instance, a neural network to represent the policy [70], [82].

#### 2.3.1.6 Proximal Policy Optimization

Proximal Policy Optimization (PPO) is a policy-based RL algorithm that aims to optimize the policy network, which consists of a neural network that maps states to actions. The algorithm optimized this network's parameters by maximizing the probability of selecting actions similar to those selected by the current policy. This algorithm contains a trust-region optimization method that prevents unstable training [83].

PPO collects a batch of transitions by running the initial policy in the environment. Then it computes the old policy probabilities with the corresponding value estimates which are computed by a different neural network, usually called value network. The new policy is computed by running the policy network with the states from the batch collected initially and is then the PPO surrogate objective function is computed by combining the ratio of the new policy with the old policy probabilities and value estimates. The objective is to maximize the surrogate objective function. This process is repeated until convergence is achieved [83].

## 2.4 Resume

This chapter provided some theoretical background on Software testing, which is important to understand the context that this thesis focuses on. It explained why automated black-box testing is more advantageous when performing fuzzy testing describing the motivation behind choosing a black-box approach. It addressed multiple types of testing and testing levels, in particular fuzzy testing methodology and a few existing successful tools that are used to conduct this kind of testing. However, it can be noted that there is a lack of valuable generic tools for RESTful API testing.

A systematic review was conducted following the PRISMA methodology, and eleven papers were considered worthy of being compared to the system being presented in this thesis. In summary, most of them still rely on randomness. Although some have already implemented RL for this task and have been successful, some do not take into consideration the combination of multiple functionalities which can lead to change of state of the API under test.

Lastly, AI was also briefly introduced to be able to explore some RL algorithms. As such, MDPs were explained because of its relationship with RL. A shortened taxonomy of RL algorithms was also presented and used to decide which algorithms could be interesting for the solution. Five different algorithms were described and explored.

# 3 FuzzTheREST

This chapter presents FuzzTheREST, the framework proposed in this thesis. In a first instance, the framework is envisioned considering its innovations and advantages compared to other previously mentioned well-known fuzzy testing systems, as well as ethical and security considerations that have to be analysed. FuzzTheREST's target audience is also discussed, which helps raising its functional and non-functional requirements. Secondly, the Software Engineering process performed is also explored by presenting use cases, an overview of the domain model and the architectural design considerations. Lastly, the RL environment implementation decisions are also detailed.

## 3.1 Conceptualisation

Software testing is a broad domain with the goal of ensuring the quality of Software, essentially by exposing it to different situations and assessing its behaviour. However, most testing types attempt to validate the SUT by exposing it to realistic inputs and scenarios, which might not be enough to validate its quality. Moreover, usually this kind of testing is made from a white-box perspective, which means that the tests are already biased, since the testers already know which user-system interactions are expected. This is still quite important for the developers to understand if their system works properly when faced with what would be an expected used input. However, it does not test most invalid input. Hence the need to validate the solution from a black-box point of view. Nowadays, this process is mostly performed by Software testers [84], and consists of interacting with the system without any knowledge of its inner workings. This way the user can try to find defects on the application. As already mentioned in Section 2.2.3, attempts to enhance this kind of testing has appeared in the form of fuzzy testing. However, typical black-box fuzzers are quite naïve as they only generate random input to be sent to the SUT until an error occurs. This is very time consuming, which makes it unfeasible, especially considering the size and complexity of modern-day Software systems [85].

In order to enhance efficiency and automate the fuzzy testing process of Software systems, AI techniques can and have been leveraged to this end. However, from a black-box perspective it is still very much convoluted, as there are multiple factors that can be related to the cause of a vulnerability. For instance, programming languages, frameworks, operating systems, hardware specifications, amongst others, can all be causes of vulnerabilities. As such, the development of a generic framework for identifying vulnerabilities in systems can be extremely challenging because of all involved factors.

FuzzTheREST, the RESTful API fuzzy testing framework presented in this thesis tries to accomplish this by proposing a tool dedicated to learning the context of each SUT. The fuzzer proposed in this thesis tackles RESTful API testing from a black-box perspective, relying on RL to solve the search and optimization problem that is finding the right input values and input combinations efficiently to uncover vulnerabilities present in RESTful APIs. The objective is to attain the highest number of vulnerabilities found in the smallest time window possible, whilst maintaining a certain level of coverage. Additionally, to enhance the value of the system and make it highly explainable, the process until finding certain vulnerabilities is fully reported and presented to the respective user of the system.

In comparison to other well-known fuzzers, this proposal also stands out for its explainability and use of RL to acquire knowledge of the API and adapt its actions to the context, without having to gather source code knowledge, as this would make the process much slower [45].

### 3.1.1 Target Audience

FuzzTheREST is a tool that focuses on finding RESTful APIs vulnerabilities, following a fuzzy testing methodology. However, it should be noted that it does not substitute the traditional white-box script-based Software testing. The framework proposed in this thesis is intended to be used by developers, testers or even security practitioners, without any prior knowledge of its innerworkings. In all these actor's roles, Software quality is of great importance for the business continuity of those who develop or use Software systems.

Considering the intended audience, the framework's development decisions should be made according to the actor's likely needs. The following section focuses on exploring the ethical and security considerations related to the use of this RL-based tool for Software testing, as this context will help clarifying the requirements of such a system.

### 3.1.2 Ethical and Security Considerations

Data is the foundation of AI algorithms [86]. It is the representation of knowledge from which these algorithms can learn to perform predictions. Hence one of the reasons why AI can be applied to such a wide variety of fields [87]. However, one must first understand the importance of data. Data refers to the representation of the understanding and interpretation of the world, which when correctly correlated permits constructing meaningful knowledge [88]. As such, data is not neutral when it comes to the context in which it is collected, methods used to gather it and the perspective of the observer. Therefore, data must be critically evaluated in order to reach correct conclusions [88].

Software systems have an enormous range of applications to various fields. As such, many times, these systems must be representative of the domain to which they are applied [89]. This can be regarding a public or private domain. The latter can be extremely problematic regarding ethical and security issues, since typically, these systems collect and instrument sensitive private data. Therefore, Software testing becomes even more important in such systems to prevent security breaches. Nevertheless, considering RESTful API Fuzzy Testing resorting to AI methods, one has to ensure the validity, anonymity and integrity of the data. Regarding security, RL raises many privacy concerns as it is capable of potentially access or manipulate sensitive information, which may lead to data breaches and privacy violation. Additionally, knowing that the main objective of FuzzTheREST is to find vulnerabilities, using a RL environment to perform this, one must consider that exploits and other undesirable behaviours of the API can lead to unauthorized access to different parts of the application. As such, it is important that the algorithm is well-designed to ensure that the agent is incentivized to find and exploit bugs and vulnerabilities, however in a simulated production environment, where the important and sensitive data are out of the fuzzer's reach, to avoid serious damage. Bias can also be an issue, as in a fuzzy testing methodology, typically, the initial data used to generate random inputs are manually entered, which can cause triggering of certain types of bugs. Another security aspect is the fact that this tool can also be representative of cyber-attack, for it performs a lot of interactions on the API with multiple input. It could be misused by a malicious actor to launch attacks, such as Denial-of-Service (DoS) or improper access to data, on the API under test. Therefore, it is crucial that security measures are implemented to prevent such misuses.

FuzzTheREST, being a testing tool that focuses on black-box type of testing, it should provide valuable information to the actor that wants to understand what has been done in the API to discover vulnerabilities. As such, the whole process should be a white-box and provide explainability and transparency. On the other hand, actors that receive the report of vulnerabilities found in the API should also consider its output to fix the Software defects identified, as these can be disruptive not only internally to those who provide the Software, but also to those who depend on it and trust its quality to perform certain tasks and provide private information.

## 3.2 Software Engineering

Considering the positioning of FuzzTheREST in several topics, it is now possible to understand the purpose of the tool from a realistic point of view. However, it now raises questions regarding its implementation. Therefore, this chapter presents the implementation decisions of the fuzzer, following good Software Engineering practices.

The following subsections describe the identified requirements, and Software design decisions, whilst providing theoretical background on the methodologies followed to fulfil this task.

### 3.2.1 Requirements Engineering

Requirements engineering is a fundamental step in the software development process. It serves as the cornerstone for creating software systems that meet the needs and expectations of users, stakeholders, and the broader context in which they operate. It involves identifying, documenting, and managing the essential functionalities and constraints that software must adhere to. Effective requirements engineering is crucial for project success, as poorly defined requirements can lead to delays and cost overruns, while well-executed processes can ensure that software not only meets but excels in meeting user expectations.

Non-functional requirements delineate the expected performance and behaviour of a software system, with a specific focus on its operational characteristics. They play a pivotal role in ensuring the software's comprehensive quality, encompassing performance, security, reliability, and usability aspects. In the scope of this project, these are the non-functional requirements:

- The implementation of the solutions should follow good Software Engineering practices, such as SOLID [90] and General Responsibility Assignment Software Patterns (GRASP) [91].
- The project must be documented in Unified Modelling Language (UML) style following the C4 Models [92] and 4+1 Architectural View [93] formalisms.
- The implementation of the solution should consider the extensibility of the project regarding the methods that can be utilized.
- The tool should provide explanation regarding the vulnerability found.

Functional requirements delineate the precise features and functionalities expected from a software system, detailing its intended behaviour and capabilities. These requirements are crucial as they serve as the architectural framework for the software's core functions, directing its development and ensuring its alignment with desired outcomes and user demands. Within the framework of this project, the following represent the functional requirements:

- The tool should permit testing any RESTful API independently of the language it is implemented on.
- The tool should provide at least Q-learning as the RL algorithm. However, more are desirable.
- The tool should allow as much configuration as possible of the algorithms available.
- The tool should allow user registration and authentication.
- The tool should restrict the number of times a user tests a certain API, to ensure that it does not constitute a cyber-attack.
- The tool should export a text file with the report of the test.

- The tool should be secured with different levels of authorization.

### 3.2.2 Practical Software Engineering Principles

During the development of the proposed solution, a strong emphasis was placed on practical Software Engineering principles that serve as the foundation for building robust and maintainable Software systems. These principles played a pivotal role in ensuring that the system not only met its functional requirements but also adhered to industry best practices. In this section, two key sets of principles, SOLID and GRASP, were briefly introduced. These principles provided valuable insights into structuring the codebase for scalability, flexibility, and ease of maintenance, which were crucial attributes for a successful Software application.

SOLID is an acronym for five object-oriented design (OOD) principles made popular by Robert C. Martin [94] in [90]. SOLID is composed of these 5 principles [95][96]:

- **Single Responsibility Principle**: A class should have one responsibility, simplifying testing and code maintenance.
- **Open-Closed Principle**: Classes should be open for extension but closed for modification, aiding in application stability.
- **Liskov Substitution Principle**: Subclasses should seamlessly replace their parent classes without altering behaviour.
- **Interface Segregation Principle**: Large interfaces should be split into smaller, focused ones to prevent unnecessary dependencies.
- **Dependency Inversion Principle**: High-level modules should not depend on low-level ones; both should depend on abstractions.

GRASP is a set of patterns to guide one understand "essential object design, and apply design reasoning in a methodical, rational, explainable way" [91]. GRASP stands for General Responsibility Assignment Software; it was first introduced by C. Larman in [91] and is composed by the following patterns [91][97]:

- Information Expert
- Creator
- Controller
- Low Coupling
- High Cohesion
- Indirection
- Polymorphism
- Pure Fabrication
- Protected Variations

Therefore, combining SOLID and GRASP principles leads to more maintainable, flexible, and high-quality software. It fosters a development approach that not only meets current requirements but also accommodates future changes and growth more effectively.

### 3.2.3 Architectural Design

This chapter covers the architectural design of this system, providing various alternatives to the problem. This will be presented resorting to UML diagrams, contemplating 4+1 Architectural View and C4 Models formalisms.

4+1 View Model is an information organization framework that aims to capture the description of Software implementation or architecture into multiple views [98]. It was introduced by P. Kruchten in [93], and it can be broken down into 5 views (Figure 11):

- **Logical View**: Is the object model of the design. In this view the system is analysed by the components that compose it.
- **Process View**: Captures the concurrency and synchronization of the design.
- **Physical View**: Maps the various Software parts into hardware, to reflects its distribution.
- **Development/Implementation View**: Describes the general organization of the Software in its development environment.
- **Scenarios View**: Illustration of the four views in terms of use cases.

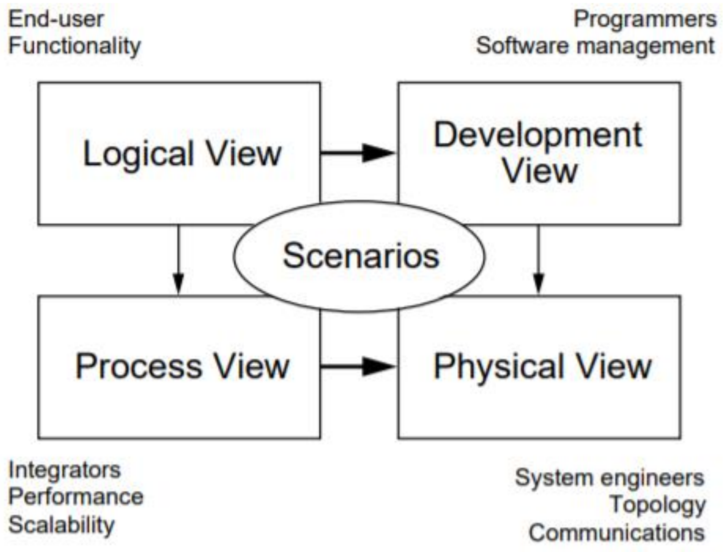


Figure 11 - 4+1 View Model [93]

C4 Model stands for context, containers, components and code, a set of hierarchical “zoom levels” that describe a system. C4 model considers structures of a Software system and the interested audience that will later use the system. In this format, it is possible to understand the system slowly and gradually by interpreting high-level diagrams first and then move to lower levels of abstraction.

C4 Model was introduced by S. Brown in [92] and this article the author presents a great analogy involving the google maps and the different levels of abstraction the map, since it can be extremely reduced and provide a general view of the world or amplified to provide lower levels of abstraction and more concrete views with finer granularity. The C4 Model can be described in these levels [92] [99]:

- **Level 1 (Context)**: Representation of the system context highest level of abstraction.
- **Level 2 (Container)**: Representation of the system's containers. It also allows to understand how these various components communicate.
- **Level 3 (Component)**: Zooms in to each container, allowing for a representation of the different components that integrate a container.
- **Level 4 (Code)**: Representation of each component implementation. This is considered the lowest level of abstraction.

When combining the two models, it is possible to achieve great interpretability and readability of the system, making use of UML diagrams, resulting in the matrix shown in Figure 12.

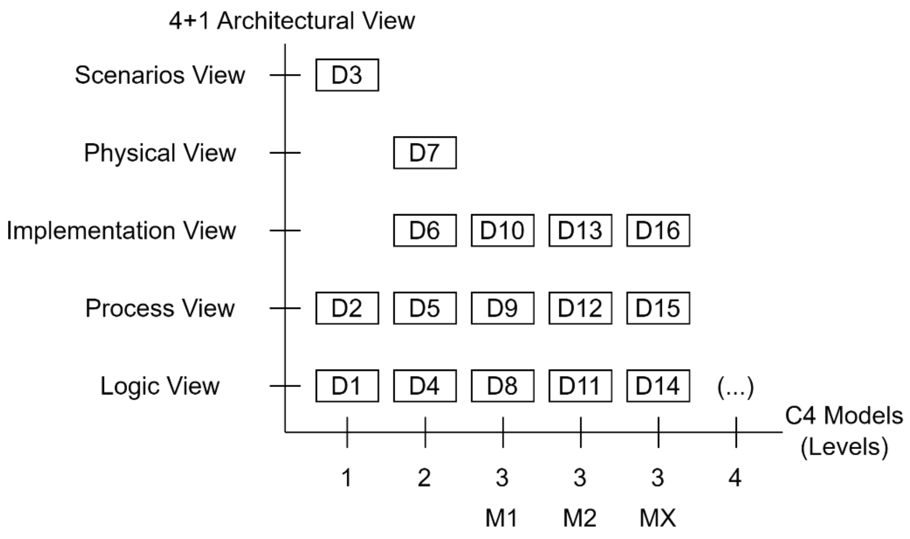


Figure 12 - C4 Models and 4+1 Architectural View Crossover Matrix

Taking into consideration the previous engineering best practices and architectural design patterns, the following subsections describe the framework proposed, examining it from two different views, logic and process in levels 2 and 3. The physical view in this context is unnecessary, as this is a framework that is supposed to be used by other users, the implementation view was disregarded, as it didn't add much value to explain the system developed. Level 1 was not introduced because it is extremely simple to a point where there is nothing to talk about, as this system by default does not depend on external systems; and Level

4 considers code, which is too specific for what is going to be presented in the following subsections.

#### 3.2.3.1 Logic View

Regarding the architecture of the system, only the logic view was considered most relevant for the development and explanation of the fuzzer.

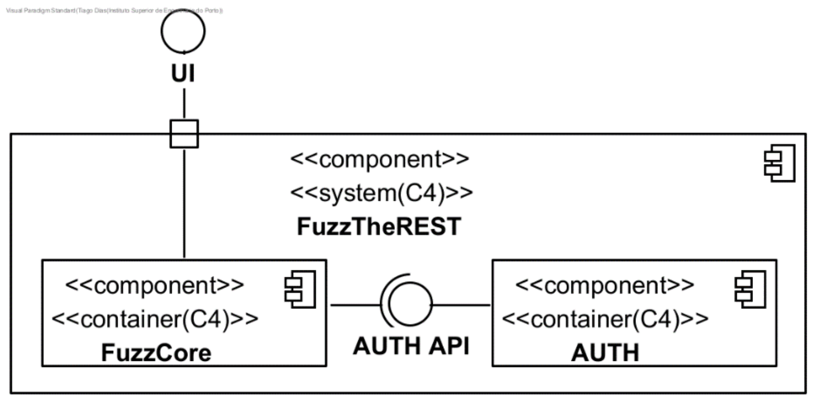


Figure 13 – Level 2 logic View of FuzzTheREST

As illustrated in Figure 13, FuzzTheREST is composed of two containers: (ii) the Authentication (**AUTH**), which is responsible for controlling the access to the tool and monitoring malicious activity; And (ii) **FuzzCore**, which is the core of the system. Its responsibility is to fuzzy test RESTful APIs, given the proper input.

AUTH is intended to contain all registered users authorized to use the FuzzCore. This Java [100] module follows a very similar architecture to FuzzCore, except it uses Springboot Framework's [101] implementation of REST. It also uses a MySQL [102] database to persist user and activity data.

FuzzCore container is intended to receive necessary input from the user, which consists of a valid OpenAPI Specification (OAS) file of the RESTful API being tested, algorithm parameters and scenario files. The scenario files consist of a combination of functionalities available in the API, which make sense to be tested in a certain order. However, this does not mean that the functionality shouldn't be tested individually, but this sequence is important to ensure maximum coverage. Without this, the system would have to perform every possible combination, which is extremely expensive computation-wise. The provided data is then used along with algorithms and data generators made available by the system to find possible vulnerabilities.

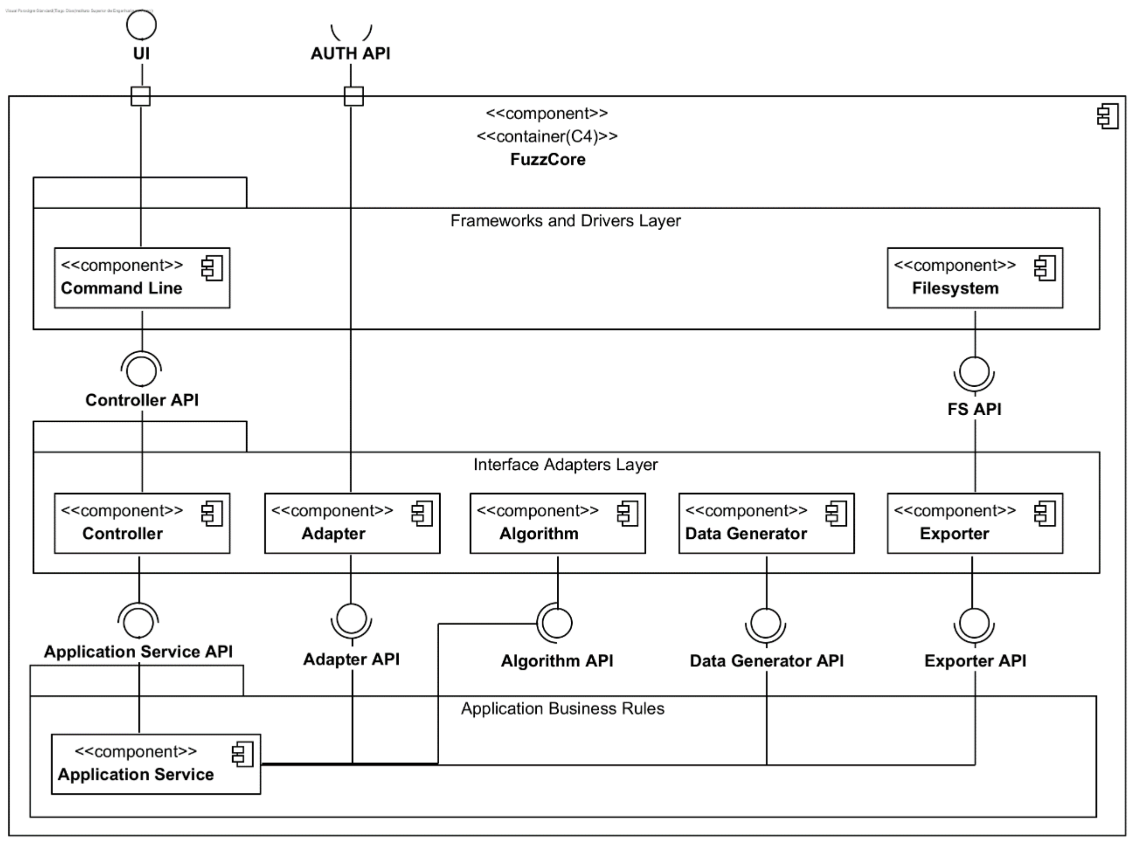


Figure 14 – Level 3 Logic View of FuzzCore

As depicted in Figure 14, the system follows an Onion Architecture, which is a design pattern emphasizing the organization of code into concentric layers with strict dependencies [103] . However, contrary to the typical four layers, this component does not consider the Enterprise Business Rules layer which contains the modelled entities and the rules of the domain. Modelling domain entities would increase unnecessary computation resources, as such it was disregarded.

The *Frameworks and Drivers Layer* contains the *Command Line* and the *Filesystem*, which are Operating System (OS) services. The first allows, in this context, the interaction with the innerworkings of the system and the second grants access to the application to operate on the filesystem. The *Interface Adapters layer* define the components which are part of the system but do not represent any business rule in particular and can have multiple implementations. This layer consists of: (i) *Controllers*, which purpose is to serve as entry points to the application and communicate with the service, (ii) *Adapters*, which can contain multiple implementations of Adapter to communicate with external APIs, (iii) *Algorithm* which may contain multiple implementations of algorithms capable of analysing the data and providing a report of vulnerabilities found, (iv) *Data Generator*, which contains the implementation of multiple data generation/mutation methods, and (v) the *Exporter*, which contains the export methods for the vulnerability report generated. At last, the Application Business Rules layer defines the application services, which orchestrate the flow of each use case. Table 3 summarizes the responsibility of each component.

Table 3 – Level 3 Fuzzer components responsibilities

| Component | Responsibility |
|---|---|
| **Command Line** | OS service for interacting with the system. |
| **Filesystem** | OS service for the system to interact with the filesystem. |
| **Controller** | Serves as entry and exit point of the API. |
| **Adapter** | Communicates with external APIs. |
| **Algorithm** | Contains multiple algorithms which can be used by the system to perform vulnerability detection. |
| **Data Generator** | Contains multiple methods utilized by the system to generate or mutate data. |
| **Exporter** | Organizes the data and defines the export methods for the final report. |
| **Application Service** | Orchestrates the flow of each use case. |

This architecture of the FuzzCore is SOLID compliant since it respects the Interface Segregation, Single Responsibility, Dependency Inversion, and Open-Closed Principle.

This system is implemented in Python programming language [104] and is considered a console application, since it uses the console to interact with the user. The algorithms component contains one algorithm, Q-learning, which is a RL algorithm. The implementation of the Q-learning algorithm utilizes Gym [105] framework. Gym is a Python library for developing and comparing reinforcement learning algorithms. It provides a variety of environments and a common interface for interacting with them, making it easy to compare different RL algorithms on the same tasks. It also allows creating custom environments and is compatible with most deep learning libraries [105]. The use of Gym in this context was considered advantageous, since it provides a common interface for interacting with the APIs, making it ideal to compare different test scenarios and compare results. Additionally, Gym permits the implementation of custom environments, which in this case allows the simulation of different states and inputs of the API. Moreover, this framework has great support for RL algorithms and a wide community. It is considered the go-to framework for RL.

NumPy [106] was also utilized, primarily for data processing tasks for its ability to provide efficient data structures and mathematical operations, ensuring optimized performance for numerical computations. Additionally, to ensure a certain level of abstraction, Textblob [107] framework was also utilized to validate and interact with the domain nomenclatures of the SUT.

Textblob is a Python library for processing textual data, providing simple APIs for common natural language processing tasks.

The communication between the system and external APIs, including the one under test, uses the requests [108] library, which is native to Python.

#### 3.2.3.2 Process View

This section presents the overall process of the proposed tool for testing an API in a summarized manner, where the RL process is summarized via class *Algorithm*, by resorting to a level 3 process view diagram produced in UML, which is on Appendix A. Notice that the user of the sequence is already the authorized user. To simplify the diagram, the authorization step was skipped, and the actor is assumed to be authorized.

As depicted in Figure 26 of Appendix A, the system starts by requesting the necessary information regarding the OpenAPI specification file, the scenarios file, the algorithm to be used along with the necessary parameters and whether an execution export is desired or not. Then the system proceeds to extract the necessary information from the OAS file, creating the data structures necessary to interact with the algorithm, which are parameterized by the user.

The algorithm then performs fuzzy testing on the functions of each scenario, with the agent interacting with the environment. The algorithm then searches for the right input combinations for the duration of iterations the user parameterized. At each iteration, to evolve the input, a series of mutations are performed on the input values, which all occur in the interaction in the interaction between the agent and the environment. During this process, the algorithm keeps track of the inputs generated, the vulnerabilities found and other metrics necessary to analyse the testing process. Additionally, if required, the system exports the testing report files of each function.

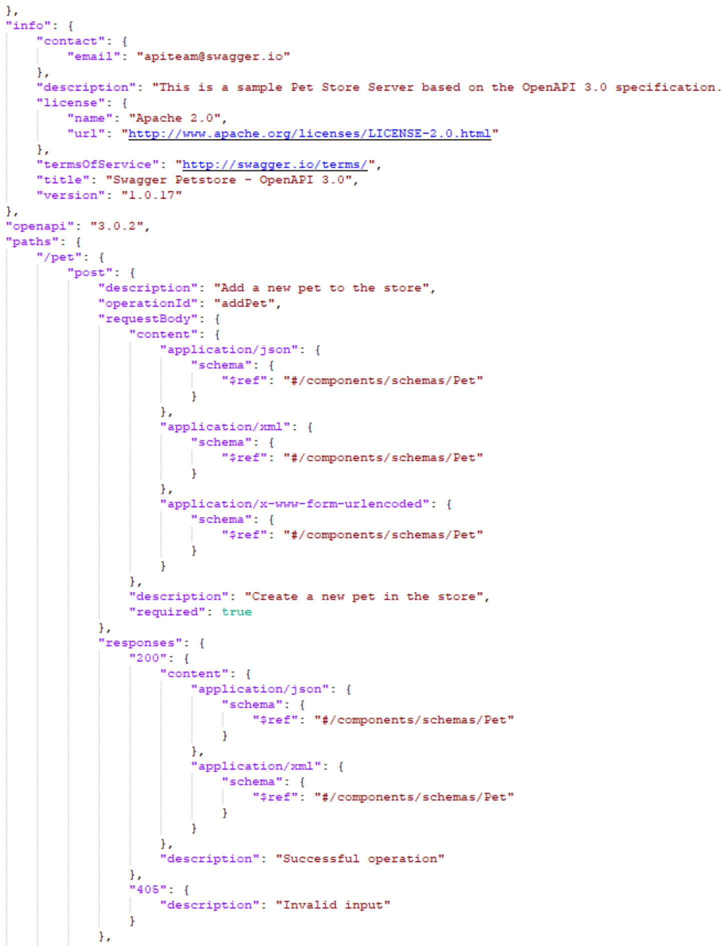
After finishing the testing of all scenarios, a file containing coverage metrics is exported in HyperText Markup Language (HTML) format.

## 3.3 FuzzCore: Reinforcement Learning Design

This chapter delineates the meticulous execution of the RL algorithm employed in FuzzTheREST. It begins by explaining the data acquisition and its subsequent processing phase. Then, it describes the environment, highlighting the reasons behind its design and characteristics. Lastly, it presents a comprehensive breakdown of the Q-Learning agent, shedding light on its inner workings and what it brings to the table.

### 3.3.1 Data Acquisition and Processing

The implemented RL works with two main instances of data: (i) the initial input data, which works as baseline to communicate with the API, (ii) and the generated/mutated data which is used to test the system.



```
},
"info": {
    "contact": {
        "email": "apiteam@swagger.io"
    },
    "description": "This is a sample Pet Store Server based on the OpenAPI 3.0 specification.
    "license": {
        "name": "Apache 2.0",
        "url": "http://www.apache.org/licenses/LICENSE-2.0.html"
    },
    "termsOfService": "http://swagger.io/terms/",
    "title": "Swagger Petstore - OpenAPI 3.0",
    "version": "1.0.17"
},
"openapi": "3.0.2",
"paths": {
    "/pet": {
        "post": {
            "description": "Add a new pet to the store",
            "operationId": "addPet",
            "requestBody": {
                "content": {
                    "application/json": {
                        "schema": {
                            "$ref": "#/components/schemas/Pet"
                        }
                    },
                    "application/xml": {
                        "schema": {
                            "$ref": "#/components/schemas/Pet"
                        }
                    },
                    "application/x-www-form-urlencoded": {
                        "schema": {
                            "$ref": "#/components/schemas/Pet"
                        }
                    }
                },
                "description": "Create a new pet in the store",
                "required": true
            },
            "responses": {
                "200": {
                    "content": {
                        "application/json": {
                            "schema": {
                                "$ref": "#/components/schemas/Pet"
                            }
                        },
                        "application/xml": {
                            "schema": {
                                "$ref": "#/components/schemas/Pet"
                            }
                        }
                    },
                    "description": "Successful operation"
                },
                "405": {
                    "description": "Invalid input"
                }
            },
```

Figure 15 – Petstore API OAS file [109]

Illustrated in Figure 15 is an example of a OAS file, which is the main data source regarding the API under test for the algorithm. This file defines the structure and capabilities of an API using the OpenAPI standard. It includes information regarding the API's endpoints, types of requests and response payloads and the authentication or authorization requirements, if any. Additionally, it also may provide examples of valid inputs to test the API. It is also in this file that we may find which status codes are expected to be outputted by the system. However, the file contains more information that what is necessary to train the RL agent to perform fuzzy testing. As such, for this context, the dimension of the data was reduced.

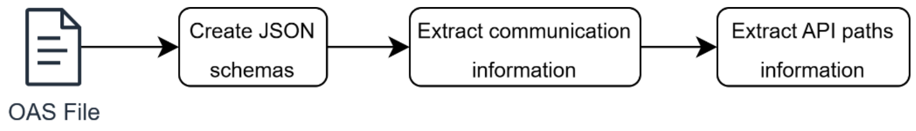

OAS File
Create JSON schemas
Extract communication information
Extract API paths information

Figure 16 - OAS File Data Extraction

The data processing phase considers the version 3.0.0 of the OAS file, which can be greatly divided into three main sequences: (i) API information, (ii) API paths and operations, (iii) and components. The first one contains information regarding the API itself, including its title, version, description, terms of service, contact information, license, and server information. The second sequence is where various endpoints are defined along with their communication specificities, such as HTTP method, parameters and request and response bodies. Lastly, the components section defines reusable components like data schemas and security schemes.

As shown in Figure 16, firstly, the components data is traversed, and the schemas are created in JSON format and stored in a list of domain schemas. Afterwards, the base Uniform Resource Locator (URL) for communication is extracted from the first section and lastly, each path in the middle section of the file is processed to create the data structures utilized throughout the system. However, in this section two different types of data structures can be defined: (i) parameters, (ii) and request bodies. The first one is usually sent in the URL query as a parameter and the second as a JSON object mainly in POST and PUT HTTP functions.

```
Function = {
        path,
        content-type,
        HTTP_method,
        input_parameters,
        input_body
}

Input_parameters = {
        name,
        location,
        schema,
        sample
}

Input_body = {
        schema,
        sample,
        schema_name
}
```

Figure 17 - FuzzTheREST Data Schema

The API paths information extracted in the last phase of OAS File Data Extraction follow the structure shown in Figure 17. Each function object contains the path of the function, the content-type, the HTTP method, the input parameters, and the input body. The input parameters specify information regarding the parameters, which includes the name of the parameter, the location where the parameter should be sent, the schema describing the datatype and information regarding if it is an array and the sample, which contains a sample value. The input body follows a very similar structure as the parameters; however, its schema specifies a JSON object with each field corresponding to a certain datatype, and the name of the schema object.

Regarding the datatypes used in this work, five different datatypes were considered: integer, float, boolean, string and byte, where some values can be arrays of a certain datatype.

Nonetheless, there's possibility for expansion. Moreover, the sample field present in both input parameters and input body data structures are an instantiated representation of the schema. This field can be randomly generated or if there is a previous sample, then this value is evolved via the mutation methods used during the testing process. The mutations methods implemented in this tool consist of nine methods adopted from successful fuzzers, such as AFL and libFuzzer [60]. These are presented and described in Table 4.

Table 4 – FuzzTheREST data generation/mutation methods [60], [110]

| Method | Description |
|---|---|
| **Bit Flips** | Randomly flips individual bits in the input data. |
| **Byte Shuffling** | Shuffles the order of the bytes in the input data. |
| **Byte Injection/Deletion** | Adds or removes random bytes, causing structural changes to the input data. |
| **Bytes Substitution** | Randomly replaces bytes for others. |
| **Truncation** | Shortens the input data by removing trailing bytes. |
| **Dictionary Fuzzy** | Substitutes certain parameters for other pre-defined ones. |
| **Arithmetic Operations** | Randomly performs arithmetic operations, such as addition, subtraction, multiplication, and division, on the input data. |
| **Random Generation** | Randomly generates input of a certain type. |

The following section describes the implementation of the RL environment taking into consideration the data specifications described.

### 3.3.2 Reinforcement Learning Environment

The RL environment implemented is a custom one since there is no similar environment for the purpose of testing Software. The approach taken is based on a chess game, however this environment is stochastic meaning that the next outcome cannot be predicted as it is extremely random. Figure 18 represents the implementation of the RL environment and agent.

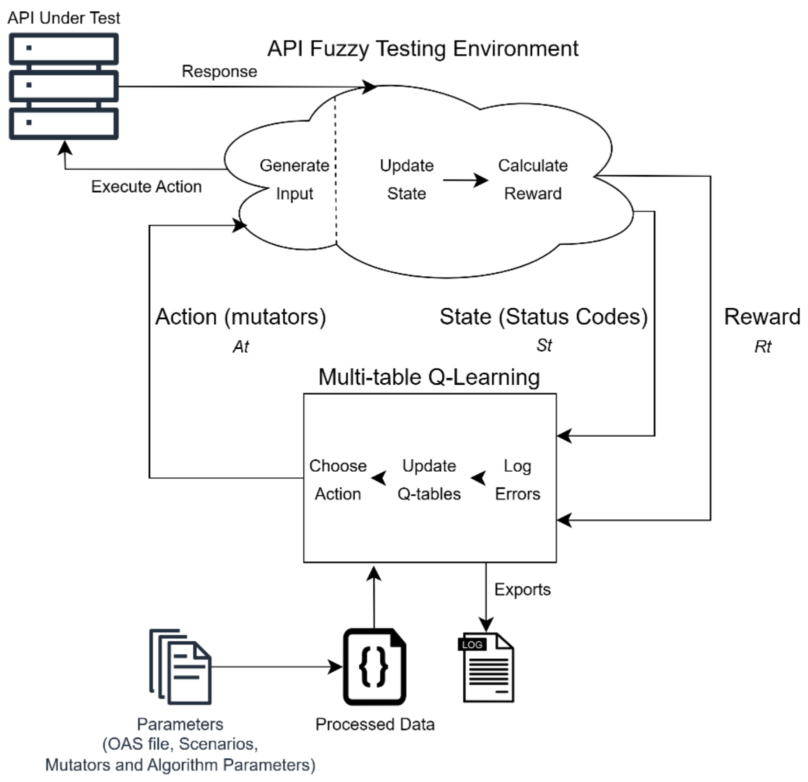


Figure 18 – Reinforcement Learning Environment and Agent Overview

To start modelling the RL environment, it is important to identify the observation and action space. The observation space refers to the set of all possible states or inputs that an RL agent can perceive from its environment. As such, in this context HTTP status codes were considered as variables for the discrete space. The different HTTP status codes are described in Table 5.

Table 5 – HTTP status codes description [111]

| Status code | Description |
|---|---|
| **1XX** | Informs that the server has received the request and processing it. |
| **2XX** | Indicates that the request has been received, accepted, processed, and successfully fulfilled. |
| **3XX** | Indicates that the request has been redirected to a different resource. |
| **4XX** | Indicates a client error or the server was unable to fulfil the request. |
| **5XX** | Indicates that the server has encountered an error while processing the request, due to an internal error or temporary condition. |

The action space refers to the set of all possible actions that an RL agent can take in its environment. Considering that the goal is to test multiple inputs against a SUT, then the actions were clearly identified as the mutation methods that can be performed. Nonetheless, this is considered a multi-discrete space, as there are different possible actions for each datatype.

The developed environment’s parameters considering the space values, communication URL and functions are all part of the parameterization of the API fuzzy testing environment. After defining the environment characteristics, it is necessary to implement the functions inherent to the environment, such as the Action, Reward, and Step. The Action function determines the action that an agent will take in a given state; The Reward function assigns a numeric value to each state and action, representing the reward or penalty associated with that state-action pair; And the Step function takes an action as input and returns the next state, the rewards and the indication that the episode is over. In the Gym custom environment implementation, these are used to define the agent’s interactions with the environment. As such, their implementation must take into consideration the data and objective of the algorithm.

The Step function which is briefly represented in Figure 26, found in Appendix A, orchestrates the interaction of the agent with the environment and contains most of the logic of the environment. In this case, this function receives the set of actions, which are the data mutators, for each data type and mutates the input values using the mutation methods, generating the new input sample. This input sample is then sent to the Action method which is responsible for communicating with the API under test and receiving the corresponding response.

The Reward function then analyses the response’s HTTP status code and assigns a reward. The rewards were implemented according to the HTTP status code: the codes in range of 1XX has 0, as reward, because it has no impactful meaning; status code in the range of 2XX and 3XX the reward is 5 points, because there was some sort of success in the communication, which is important to ensure that the system is being tested, and so the agent might want to explore that input or close values until a vulnerability is found; in contrast, the 4XX code range is rewarded -20 points because a response with this status code means that the request is not being correctly made and so the API’s innerworkings are not being fully tested. Nonetheless, hitting some of these status codes might be important as they may reveal defects; lastly 5XX is the code range the agent wants to achieve and so when reached the agent is rewarded 10 points.

Considering the implementation of this environment, the agent leverages the environment to learn characteristics of the API under test. The following section describes the agent chosen as well as its implementation characteristics.

### 3.3.3 Multi-table Q-Learning Agent

An agent is an algorithm that interacts with an environment to learn and make decisions to maximize a cumulative reward. The agent's goal is to learn a policy that maps observations from the environment to actions. This policy guides the agent's decision-making process, helping it choose actions that lead to favourable outcomes over time.

Regarding the algorithm used for this task, a model-free Multi-table Q-Learning algorithm [112] was implemented as it is fit for the task at hand, since it does not require prior knowledge of the domain and deals well with stochastic environments. In this algorithm, the state-action observation pairs are stored in Q-tables. However, because the observation space of the environment is multi-discrete, the agent has separate Q-tables for each datatype. Considering how uncertain and patternless the domain can be and to learn the best mutation methods, an exploration-exploitation policy [113] should be considered to ensure that the algorithm is capable of learning each API's behaviour. The objective is to be able to learn the best action to perform at each iteration, but at the same time be able to explore other methods and inputs. Therefore, an epsilon-greedy policy [114] with an exploration decay was employed.

Epsilon-greedy policy balances exploiting the current knowledge with exploring new actions. It selects an action based on the highest Q-value with a high probability, and selects a random action with a low probability, allowing the agent to exploit its current knowledge, whilst exploring new actions according to the Q-table [115]. The exploration decay ensures that in an initial phase, the algorithm is exploring the space, and later it starts exploiting points of failure. Overtime, the agent should be able to learn which mutation methods should be utilized in each iteration to best know the API under test and its vulnerabilities.

#### 3.3.3.1 Agent Training Pipeline

In order to find vulnerabilities in systems, first the agent must adapt and understand the environment. Only then can it exploit the SUT to find vulnerabilities.

Figure 27 in Appendix B depicts the training process of an agent in a more detailed manner than previously presented, describing the interaction between the agent and the environment during this process. The agent is trained during a parameterized amount of training episodes, and during each episode, the agent interacts with the environment, registering the outcome of the interaction, updating the Q-values of each Q-table. The Q-values represent the success of state-action pairs and are calculated using the Bellman Equation. Afterwards, the agent relies on the defined policy to choose the new mutation method that should be used in the next episode. In each episode, the agent has a user-defined number of steps to take. This value defines when an episode ends if reached, otherwise the episode may end sooner if the agent reaches an HTTP status code in range 5XX.

Nonetheless, generating random data and sending it to an endpoint may not always be sufficient for an agent to be well trained. One must consider the dependencies that exist between functionalities of the same system, since disregarding them may result in an agent with an incomplete training.

To solve this issue, a mechanism for sharing dependencies between functions was implemented. Since each agent only tests one function, if the function being tested consists of a creation function, then the agent stores the unique identifiers that are generated so that other agents that may require that information have access to it. The implementation of this mechanism relies on substring identification and word inflection methods available in Textblob to singularize the components names and respective identification field. This method works both to store the identifiers and to access them. Even though this functionality was only implemented for unique identifiers, it is extensible to include a wider variety of fields.

## 3.4 Resume

In the initial section of this chapter, the conceptualization of FuzzTheREST was introduced, the target audience was examined, and ethical and security considerations were explored. This initial exploration was imperative in establishing the intended direction for this tool. Once the purpose and objectives of the tool were clearly defined, the focus shifted towards constructing and analyzing the solution from a software engineering perspective. Within this chapter, the requirements, encompassing both functional and non-functional aspects, were elucidated, building upon the conceptualization and earlier considerations. Emphasis was placed on the practical principles, such as SOLID and GRASP, which play a pivotal role in ensuring the robustness of the system.

Furthermore, architectural design discussions were delved into, introducing the theoretical foundations of the 4+1 and C4 models. These models enabled a more detailed breakdown and analysis of the logic view and process view, providing a comprehensive understanding of the system's structure. In this context, the tools and frameworks employed in this work were also presented.

Subsequently, the chapter transitioned to the design of the reinforcement learning component. Insights into the methods used for data acquisition and processing were provided, thoroughly reviewing the input structures, data types considered, and the mutators implemented in the final solution. Additionally, the foundation was laid by describing the custom RL environment developed specifically for API fuzzy testing. Within this chapter, the Multi-table Q-learning Agent was introduced, elucidating its training pipeline and its interaction with the environment.

# 4 Demonstration

To demonstrate the suitability and validity of the proposed solution, this chapter presents two case studies which utilize as testing subject two RESTful APIs: Petstore API [109] and the Human Behaviour Analyzer module (HBA) [9], [10] which integrates the CyberFactory#1 project [116], [117]. The following subsections describe the selected RESTful APIs, the behaviour of the fuzzer and the results obtained. These are then discussed to evaluate the value, the performance, and the achieved objectives, but before delving into the those, it's important to establish the common preparations made for each case study.

Considering that the goal of this tool is to find vulnerabilities in a production-ready REST APIs, one must prepare a suitable testing environment. To replicate authentic operational conditions, both APIs were setup in a simulated production environment. The Petstore API was compiled using Maven [118] for generating a production build, whilst HBA, the Python application, was deployed in multiple scripts. Subsequently the two systems were deployed through Docker [119] in a containerized manner, to ensure the veracity of the testing outcomes. A comprehensive evaluation of the testing tool's performance necessitated a robust measure of code coverage. As such, a code coverage agent was employed in both APIs. JaCoCo [120], functioning as a Java Virtual Machine (JVM) agent, facilitated the dynamic capture of code coverage metrics of the Petstore API. However, in the Python application HBA, to the best of the authors knowledge, there is yet not tool nor framework capable of collecting live code coverage, other than by analysing test files, which is not the goal. One additional similar aspect is the algorithm’s parameters that are used across both the Petstore API and HBA. These parameters play a vital role in shaping the testing scenarios and provide a consistent basis for evaluating the effectiveness of the testing tool. Table 6 describes the parameters considered for the execution of the fuzzer.

Table 6 - Case Study Algorithm Parameters

| Parameter | Value |
|---|---|
| **Integer mutation methods** | bit_flips, byte_shuffling, bytes_substitution, arithmetic_addition, arithmetic_subtraction, arithmetic_multiplication, arithmetic_division, random_generation, dictionary_fuzzy |
| **Float mutation methods** | bit_flips, byte_shuffling, bytes_substitution, arithmetic_addition, arithmetic_subtraction, arithmetic_multiplication, arithmetic_division, random_generation |
| **Boolean mutation methods** | bit_flips, byte_shuffling, random_generation |
| **Byte mutation methods** | bit_flips, byte_shuffling, byte_injection, byte_deletion, bytes_substitution, truncation, random_generation |
| **String mutation methods** | bit_flips, byte_shuffling, byte_injection, byte_deletion, bytes_substitution, truncation, random_generation |
| **Number of max. steps** | 5 or 10 |
| **Number of episodes** | 50 or 100 or 200 or 500 |
| **Exploration rate** | 1 or 0.8 |
| **Discount factor** | 0.9 |
| **Min. exploration rate** | 0.01 |
| **Max. exploration rate** | 1 |
| **Exploration decay rate** | 0.01 |

## 4.1 Petstore API Case Study

The Petstore API is an illustrative Java-based RESTful API intricately developed using the Spring framework [101], built on the OAS 3.0.0. It serves as a model example of a web service, designed around the concept of a pet store business. This API encapsulates quintessential layers characteristic of information systems, encompassing infrastructure, data, application services, and domain and is often utilized for educational purposes, providing insights into web service development and testing.

The API's functionalities are organized into three core categories, each centered around specific entities.

The Pets section manages operations related to pets, forming the backbone of the pet store's offerings. This category contains the functionalities mentioned in Table 7

Table 7 - Petstore API Pets Methods

| Function Name | Description |
|---|---|
| **AddPet** | Adds a new pet to the store |
| **UpdatePet** | Updates an existing pet |
| **UpdatePetWithForm** | Updates a pet in the store with form data |
| **FindPetsByStatus** | Fetches pets by their status (available, pending and sold) |
| **FindPetsByTags** | Fetches pets by their tags |
| **FindByPetId** | Fetches a pet by its identification number. |
| **UploadImage** | Uploads an image of the pet |
| **DeletePet** | Deletes a Pet |

The store section enables the placement and tracking of orders for pets, facilitating transactions, and the methods available are the ones presented in Table 8

Table 8 - Petstore API Store Methods

| Function Name | Description |
|---|---|
| **PlaceOrder** | Places an order for a pet |
| **GetOrderById** | Fetches a purchase order by its identification number |
| **GetInventory** | Fetches pet inventories by their status (available, pending and sold) |
| **DeleteOrder** | Deletes a purchase order |

The last category, Users, permits the registration of users to interact with the Petstore. This category contains the methods described in Table 9.

Table 9 - Petstore API Users Methods

| Function Name | Description |
|---|---|
| **CreateUser** | Creates a new user |
| **CreateUsersWithArrayInput** | Creates a list of users with given input array. |
| **CreateUsersWithListInput** | Creates a list of users with given input list |
| **Login** | Logs a user into the system |
| **Logout** | Logs out the current logged in user session |
| **GetUserByName** | Fetches a user by its name |
| **UpdateUser** | Updates a existing user |
| **DeleteUser** | Deletes a user |

The Petstore API includes multiple Create, Read, Update and Delete (CRUD) functionalities revolving around the main entities, some of which contain different customizable parameters, especially fetching operations. To holistically gauge our testing tool's effectiveness, three distinct testing scenarios were crafted. These scenarios targeted CRUD operations involving pets, users, and for last orders, ensuring thorough exploration of diverse API functionalities and the potential vulnerabilities they might harbour.

This API is also targeted by many literatures works for software testing and vulnerability discovery [42], [121]–[123]. As such, it can be considered a good candidate for testing.

### 4.1.1 Execution and Results

Harnessing the parameters detailed in Table 6, a suite of 18 agents were deployed and trained. Out of all combinations of parameters, the global best combination was the one where the maximum number of steps per episode was 10, for 500 episodes with a learning rate of 1. However, from the reports extracted from the process it is possible to see that some agents reach a plateau much sooner, meaning that perhaps for those agents, other parameters would fit best.

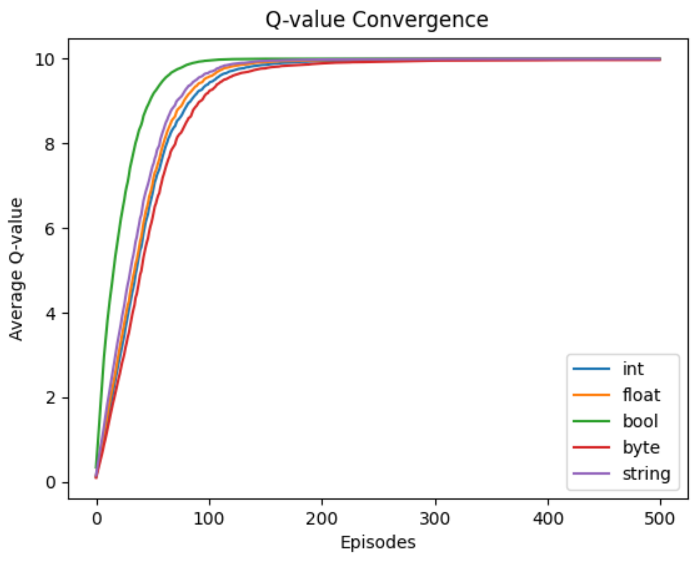


Figure 19 - Q-Value Convergence of addPet Function

An example of this is the agent of the function ***addPet***, where it is clearly seen reaching a plateau at the 200th episode, as shown in Figure 19.

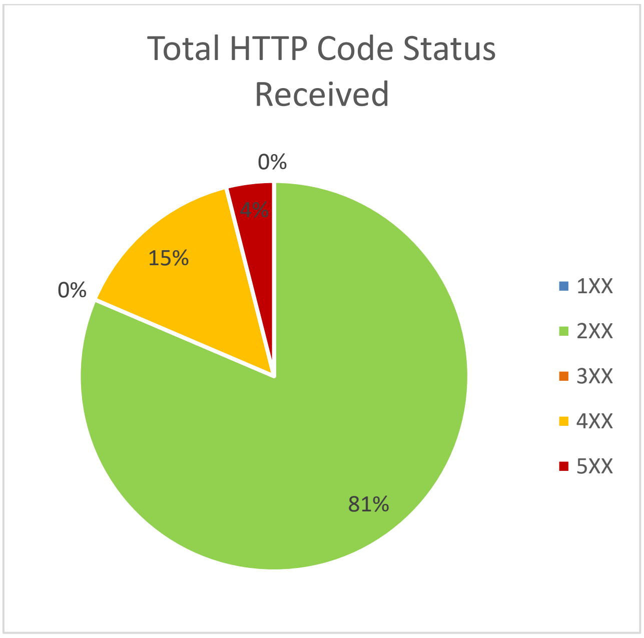


Figure 20 - Total HTTP Code Status Received

A compelling pattern emerged in the API's responses, offering intriguing insights. As shown in Figure 20, during testing, a remarkable proportion of requests—52531—yielded 2XX status codes, signifying successful outcomes. Notably, contributions to this success stemmed from functions like ***addPet***, ***createUser***, and others, underpinning the robustness of these particular functionalities. Conversely, the testing process elicited 8139 instances of 4XX status code responses. Predominantly originating from functions such as ***deleteUser***, ***findPetsByStatus***, and ***placeOrder***, these errors underscored potential issues within these areas. The exact contributions of each function can be found in Appendix D.

When considering the learning process of the agents, insights can be drawn from the analysis of the state visits and Q-value convergence plots provided in section 0. These visual representations provide a clear illustration of the learning progress exhibited by each agent. Notably, some agents commence their training with a significant proportion of 4XX status codes, gradually transitioning to an increased occurrence of 2XX and 5XX codes. This evolution signifies the effectiveness of automated requests in the learning process. Furthermore, in light of the fact that all identifiers assume integer values, a deeper examination of the chosen mutation method for integers corroborates this observation. This validation is particularly evident in the action distribution plots, reinforcing the assertion that the automated requests are indeed yielding productive outcomes. However, this is not always the case. In specific, the ***findPetsByStatus*** was the only function where an agent was unable to learn anything from it, having 100% 4XX hit ratio. This happens because the status parameter only had three possible string values, which is not easy for the agent to learn, since most inputs are randomly generated or mutated. The plots in Appendix C.4 show the randomness and continuous sense of exploration of the agent towards this function.

During the tool's testing endeavours, a subset of requests—2549—culminated in 5XX status codes, predominantly tied to functions like ***deleteOrder***, ***getInventory***, and others, which can be found in Appendix D. While this volume of errors raises concern, it's important to note that not all instances represent distinct vulnerabilities. A meticulous analysis of API responses during testing yielded insights into the nature of these errors and their potential implications. Even though 2912 requests were replied with a 5XX status code, this doesn't mean that there are all this number of vulnerabilities. Some of them are the same kind of vulnerability, just with different input. After a thorough analysis of the report document of each function which provides the responses given by the API during testing, 7 different vulnerabilities have been identified, these are described in Table 10. This highlights the significance of the report document and demonstrates FuzzTheREST's capacity for providing explanations.

Table 10 - Petstore Vulnerabilities

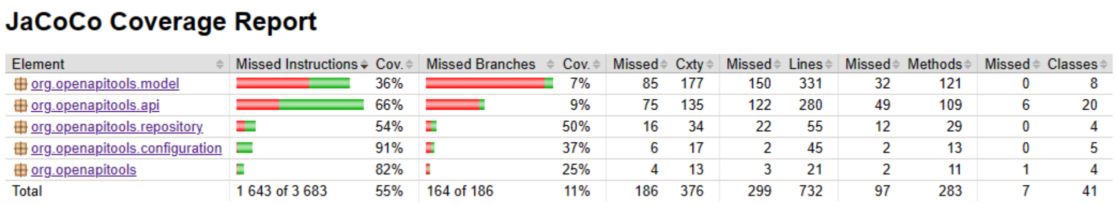
| Vulnerability | Status Code | Description | Faulty framework |
|---|---|---|---|
| **For input error** | 500 | The error message for this vulnerability did not provide much detail, but the system may be trying to convert a string input to a number. | Not applicable |
| **URL contains potential malicious content** | 500 | Issue related to input validation or security checks within the application. | Spring boot security framework |

| | | | |
|---|---|---|---|
| **Broken surrogate pair** | 500 | Surrogate pairs are used in Unicode encoding to represent characters outside the Basic Multilingual Plane (BMP). In this case, the first character and the second character form a surrogate pair that is considered illegal. The error message shows that the object which was previously recorded successfully is unable to be fetched [124]. | com.fasterxml.jackson.core |
| **Invalid white space character** | 200 | Invalid white space character makes the fetch of certain objects previously recorded impossible. In eXtensible Markup Language (XML) 1.1, such characters could be represented as character entities, but in JSON, they are not allowed [124]. | com.fasterxml.jackson.core |
| **Unmatched second part of surrogate pair** | 500 | In Unicode encoding, characters outside the BMP are represented by two 16-bit values called surrogate pairs. The first value is the high surrogate, and the second value is the low surrogate. These two values should always appear together to represent a single character. The error message suggests that there is a second part of a surrogate pair without a matching first part. | com.fasterxml.jackson.core |
| **Fetch operation unsuccessful (GetInvetory)** | 500 | No error description provided. However, by analysis of the source code, the function attempts to fetch | Not provided |

| | | | |
|---|---|---|---|
| | | the number of pets for each status. However, there is no validation for the status with which the pet is recorded in the system. As such, when it finds an unexpected status, the code fails to execute, leading to an internal server error. | |
| **Delete operations** | 200 | Even though no error was raised from the delete operations, it was verified that no warning or error was raised when attempting to delete an absent object. | Not applicable |

Moreover, the Petstore API vulnerabilities documented in the literature [122] were also identified by FuzzTheREST tool. However, unlike FuzzTheREST, most scientific works that present fuzzers or other testing tools, do not describe the errors found and most times only focus on the number of times the error code 5XX is received [42], [123], [125], [126].

Despite having made a great number of requests to the API functions and attain good results in multiple functions, one must understand if the Software was thoroughly tested or not. It is important to analyse how much of Software was tested.



JaCoCo Coverage Report

| Element | Missed Instructions | Cov. | Missed Branches | Cov. | Missed | Cxty | Missed | Lines | Missed | Methods | Missed | Classes |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| org.openapitools.model | | 36% | | 7% | 85 | 177 | 150 | 331 | 32 | 121 | 0 | 8 |
| org.openapitools.api | | 66% | | 9% | 75 | 135 | 122 | 280 | 49 | 109 | 6 | 20 |
| org.openapitools.repository | | 54% | | 50% | 16 | 34 | 22 | 55 | 12 | 29 | 0 | 4 |
| org.openapitools.configuration | | 91% | | 37% | 6 | 17 | 2 | 45 | 2 | 13 | 0 | 5 |
| org.openapitools | | 82% | | 25% | 4 | 13 | 3 | 21 | 2 | 11 | 1 | 4 |
| Total | 1 643 of 3 683 | 55% | 164 of 186 | 11% | 186 | 376 | 299 | 732 | 97 | 283 | 7 | 41 |

Figure 21 - JaCoCo's Code Coverage Report

The JaCoCo's code coverage report depicted in Figure 21, indicates that in total, 55% of code coverage was achieved throughout the testing process. Naturally, due to the nature of the fuzzer, it is possible that not every execution path is reached, which of course reflects on the code coverage achieved. Nonetheless, after analysing the source code, it was noticed that several classes that have defined methods, which are included in the coverage calculation are never utilized, meaning that such code will never be executed nor tested. As such, if the unused code was to be removed, the coverage would increase significantly, as there are many methods that contain multiple instructions that are never traversed.

## 4.2 CyberFactory#1 Case Study

CyberFactory#1 was a project that aimed to enhance the optimization and resilience of the Digital Factory and Factories of the Future (FoF) through the design, development, integration, and demonstration of a set of key enabling capabilities. The project addressed the needs of various industries such as transportation, automotive, electronics, and machine manufacturing. It proposed of the capabilities to optimize the efficiency and security manufacturing process, as well as addressing cyber and physical threats and safety concerns. The project also aimed to solve not just the technological challenges of Industry 4.0 but also the technical, economic, human and societal dimensions. The project achieved this by delivering realistic digital models of FoF, developing key technology bricks for the optimization of the manufacturing cycle, and addressing the need for enhanced resilience of FoF [116].

GECAD was a partner of the project and contributed by proposing a multi-domain security awareness tool for FoF, incorporating three different domains: (i) energy, (ii) human behaviour, (iii) and network. In the system developed by the authors, each domain is represented by a module, which has its own characteristic and responsibilities, that ultimately monitor those sectors and raise alerts for each of them. The alerts of each domain are gathered in an intelligent correlator [127], and are correlated to infer if the factory is suffering a safety breach.

In particular, the human behaviour domain is represented by the HBA component. It is responsible for capturing the emotions of the shop floor operators. The component is composed of two distinct containers: (i) Camera Application (ii) and Emotion Detector. Figure 22, displays the architecture of this component.

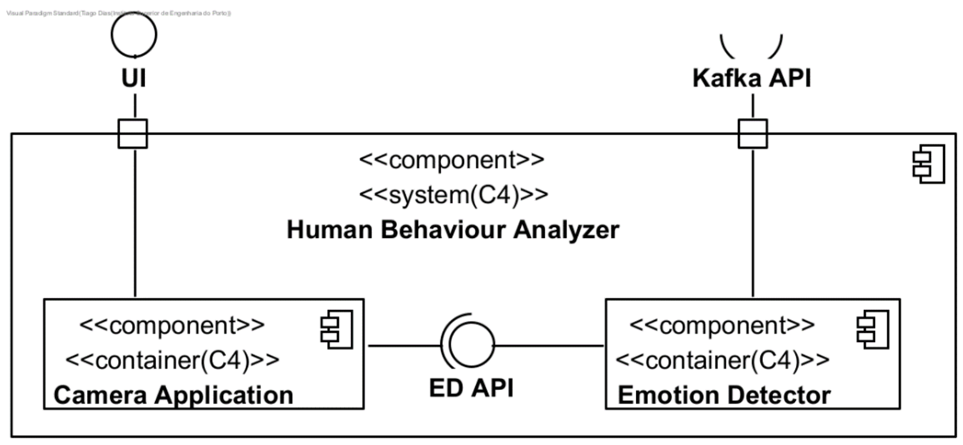


Figure 22 – Human Behaviour Analyzer architecture [9]

The Camera Application is a standalone user Interface (UI) container, with the responsibility of capturing frames of the worker's face and displaying information provided by the Emotion Detector. The Emotion Detector is a ML-powered container, which is responsible for receiving frames via the Camera Application, processing the data and performing emotion recognition, considering the seven emotions of Paul Ekman's Model of Basic Emotions [128], based on the facial expression of the workers. These would then be sent over a Kafka[129] topic to the intelligent correlator and the worker would receive the feedback via the Camera Application.

The Emotion Detector is considered a RESTful API in this component's architecture, possessing its own OAS file. Therefore, it is a great candidate for conducting experiments to validate the proposed solution, since it also integrates DL which could raise much more interesting different vulnerabilities.

### 4.2.1 Execution and Results

Once again, the algorithm was setup with the parameters detailed in Table 6. However, this time, because the system only contains one function, which is to analyse an image, find a person's face and detect its emotion, only one agent was created and tested. Out of all combinations of parameters, the global best combination was the one where the maximum number of steps per episode was 5, for 100 episodes with a learning rate of 1. The increase of these parameters led to much higher computation times, as this API calls are extremely complex computation-wise.

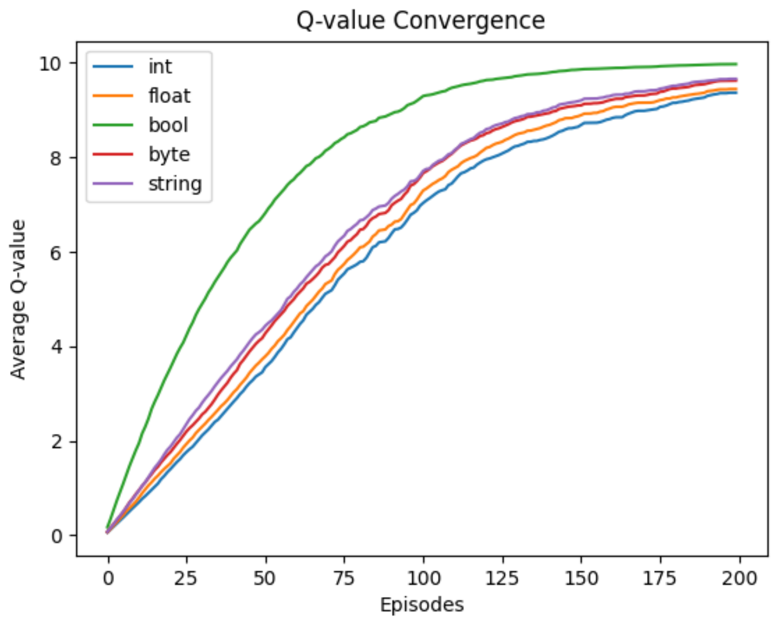


Figure 23 – DetectEmotion Agent's Q-Value Convergence

The evaluated method exclusively employs a single byte-encoded frame parameter transmitted through a JSON object. As depicted in Figure 23, the agent can be seen starting to stabilize at the 200th episode, which could be attributed to the nature of the input data. The API is set to receive an array of bytes, regardless of its order sequence, size, or byte values, which should represent a frame. However, regardless of these attributes, a system will always interpret the bytes array as if it was a byte encoded frame.

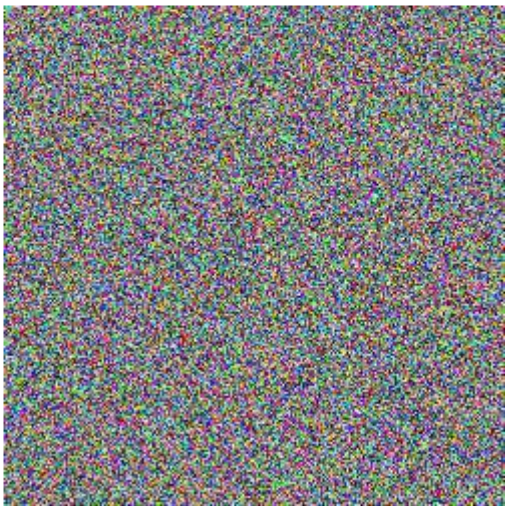


Figure 24 - Randomly Generated Test Frame

Since the data is randomly generated/mutated by the fuzzer, the resulting byte information can be a meaningless, leading to a randomly generated nonsensical frame, such as the one shown in Figure 24. Thus, even when the input generated is a nonsensical frame, the API consistently processed it into frames for subsequent analysis.

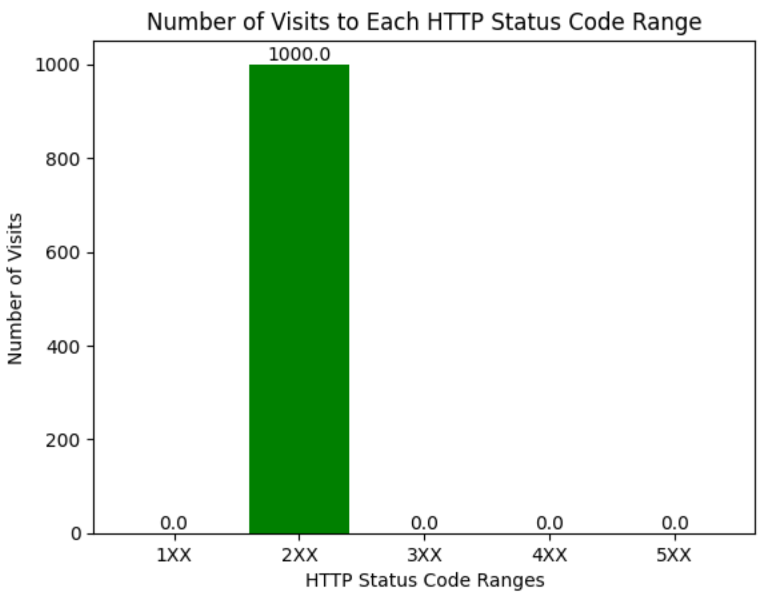


Figure 25 - DetectEmotion Agent's State Visits

Consequence of this agnosticism, the fuzzer is consistently obtaining 2XX status codes from the API, as evidenced in Figure 25, consequence of the lack of meaning in the images, when the API applied facial detection algorithm, it resulted in responses indicating the absence of detectable individuals within the processed images. This could be justified for the way that the function is implemented. For instance, if the function were to receive a base64 encoded string image, perhaps it would be more prone to errors and having vulnerabilities.

## 4.3 Discussion of Results

The results obtained from the two case studies provide valuable insights into the effectiveness of FuzzTheREST. This tool demonstrates its capacity to understand the intricacies of different RESTful APIs and adapt to varying API contexts efficiently. Notably, the tool's usage does not impose significant time constraints on the testing process, rendering it a valuable asset for Software testing endeavours. Additionally, the reports extracted from the tool confer it great explainability.

In terms of the primary objectives, the authors found that FuzzTheREST exceeded expectations by successfully identifying both visible code errors and hidden vulnerabilities. These vulnerabilities, often elusive and challenging to uncover through traditional scripting, were effectively revealed by the tool. When examining the distribution of status code responses, a substantial number of responses fell into the 4XX category. However, exceptions were noted, particularly in functions where agents encountered limitations associated with predefined values. A detailed examination of each agent's report revealed a discernible learning curve. After encountering initial 4XX responses, the agents displayed a propensity to transition towards obtaining more 2XX responses, occasionally interspersed with 5XX status codes.

Beyond the immediate implications of each status code, it is vital to acknowledge the significance of eliciting diverse status code responses. Different codes often signify the exploration of distinct system paths within the SUT, contributing to a more comprehensive evaluation and increase in coverage. Additionally, the results presented show the duality of values of the status codes. For instance, in the results achieved, the status code 500 does not necessarily mean that a vulnerability exists, but instead, an analysis into the root cause should be made. On the other hand, the status code in 2XX usually means that the operation was successful, when the results show that this is not always the case. Sometimes a data leak or a broken access control can cause an erroneous 2XX status code.

Regarding the parameterization chosen, the authors believe that the parameters chosen were ideal for both APIs, however, this may not always be the case. For instance, a more complex system will certainly have more execution paths to explore. The agents will take longer to get to know better longer and more diverse paths. On the other hand, systems with complex functionalities could also pose threat to the time it takes to test a system. Therefore, parameters involving the exploration rate and the number of episodes and steps should always be fine-tuned to each case.

In the context of fuzzing tools, FuzzTheREST distinguishes itself by leveraging RL to uncover vulnerabilities. This approach sets it apart from tools like AFL and LibFuzzer. While AFL and LibFuzzer are widely recognized for their effectiveness in traditional fuzzing tasks, they do not possess the adaptability and learning capabilities inherent in FuzzTheREST. Moreover, FuzzTheREST consistently achieves substantial code coverage, further emphasizing its ability to learn intricate API interactions and identify various vulnerabilities, including those that remain concealed.

## 4.4 Resume

This chapter encompassed a comprehensive examination of FuzzTheREST's demonstration and results on the two defined case studies. Firstly, the details regarding the parameters employed for API testing, along with the technologies used to prepare the APIs were described. Afterwards, each API under test was described along with their functionalities. After execution, the results obtained in both APIs regarding the learning process of the agent, vulnerabilities found, distribution of states per function and code coverage achieved were analysed. Lastly, a discussion regarding the results achieved took place to discuss the results obtained and how these reflect the success of the work developed.

Then, the Petstore API and its available functionalities were described, and the results achieved by the testing agents and results were presented and conclusions were drawn from those. Afterwards, the same applied to the CyberFactory#1 module, HBA, case study. This chapter functioned as the foundational cornerstone of the thesis, providing a structured framework for the subsequent exploration of the research contributions and their implications.

# 5 Conclusion

This summarizes the results obtained, presenting the main conclusions drawn from this thesis. In this chapter, the identified research questions and objectives are analysed to understand if this work answers them properly.

## 5.1 Summary of results

As Software systems increase in size and complexity, so does the need for testing. This is the result of Software systems gaining more attraction and becoming more impactful of people's daily life. A Software that lacks quality can be extremely disruptive in many harmful ways. As such, to evaluate and enhance Software quality, Software testing is performed. Many types of testing have been adopted by developers and are performed to ensure the quality of a Software. However, these tests, which are manually developed and usually considered white-box, only evaluate the requirements of the domain. It is necessary to evaluate the system with not so expected input, which can be designated as faulty input. To perform this in a white-box manner would be extremely inefficient, as it would take a lot of time since the input search and combinatorial space is so broad and the input created for each test case would be already biased. As such, it is important to approach this problem from a black-box perspective and use intelligent methods to reduce the input universe.

This thesis presents FuzzTheREST, which is an intelligent RESTful API black-box fuzzer. The tool performs fuzzy testing to evaluate the quality of RESTful APIs, generating inputs in an intelligent manner resorting to RL to guide the input mutation methods utilized to find vulnerabilities and get acquaintance with the API behaviour. From the systematic review conducted in this work, FuzzTheREST stands out for its attempt to guide the input being generated by trying to understand the underlying code resorting to RL, for its adaptability to a wide range of APIs following the OAS 3.0.0 file format for establishing communication with the APIs under test and for wide range of configuration that allow its setup according to the respective use case and providing transparency and explainability throughout the process, whilst taking ethical and security considerations to guide the implementation of the tool, which is something that to the best of the authors knowledge has not yet been made.

The case study conducted on the two APIs, Petstore API and Cyberfactory#1 module, showed the custom environment and Q-Learning agent developed were successful for the task of fuzzy testing, vulnerability discovery and documentation. The tool was able to learn the context of the API, uncover vulnerabilities, and achieve a great live code coverage considering that parts of the APIs were implemented as functions but never called for any instruction, increasing the amount of code that is left unseen and lowering the code coverage. The study conducted also showed that there is a certain duality to HTTP status codes as vulnerabilities were found in the range of 2XX and some status codes 5XX may not be necessarily vulnerabilities, or even Software defects.

## 5.2 Objectives Achieved

Table 11 provides a summary of the objectives outlined in Section 1.3, which have been successfully accomplished.

Table 11 – Objectives fulfilled

| Objective | Description | Achieved |
|---|---|---|
| **O1** | Investigate the current state of the art of automated Software testing. | ✓ |
| **O2** | Investigate the current state of the art of constrained input generation | ✓ |
| **O3** | Identify the current flaws of automated Software testing tools and proposed solutions | ✓ |
| **O4** | Propose a RL environment in the context of Software testing | ✓ |
| **O5** | Design and implement a RL-based automated Software testing tool equipped with methods for fuzzy black-box testing | ✓ |
| **O6** | In a case study, evaluate the developed work in a well-known public API and in a private API which integrates a GECAD project for vulnerability discovery | ✓ |

## 5.3 Research Questions Answered

In Section 1.3 four research questions were raised, and answers can already be provided based on the achieved objectives. The main conclusions that can be drawn from them are the following:

### 5.3.1 RQ1 - What are the main flaws of current automated Software testing applications?

Most automated Software testing tools perform exhaustive testing to achieve their results from a black-box perspective. It is beneficial when it comes to the number of tests that are performed. However, most of them wont lead to any interesting result regarding existing vulnerabilities of the SUT. Most of them still lack intelligent input generation. To the extent of what has been research in this field, some tools also do not provide as much metrics and user feedback content as one would need to evaluate the results achieved in order to fix vulnerabilities found. Additionally, none of them tackle the ethical and security issues that are inherent to a fuzzy testing tool which focuses on heavily consuming resources from APIs under test. Moreover, most tools disregard application scenarios, which combine multiple functionalities. The combination of different functionalities may also lead to state changes of the API which can raise new vulnerabilities.

RQ1 is mainly addressed in the systematic review of Chapter 2.

### 5.3.2 RQ2 – Which specifications/methods can be utilised to avoid exhaustive web API testing?

Research works typically rely on OpenAPI specification to establish a base line with the API regarding available paths and input expected. Additionally, these files may contain examples which can be utilized to diminish the search space. Moreover, scenario file can be constructed to be able to reduce the number of permutations between methods in order to achieve more code coverage. Additionally, RL and metaheuristics can be utilized to diminish the input universe.

RQ2 is mainly addressed in the systematic review in Chapter 2.

### 5.3.3 RQ3 – Which are the methods utilised for test/input generation?

Most methods utilized for input generation rely heavily on random generation, API under test response data, public knowledge bases, custom data generators and API specification files. More recently, some intelligent methods have also appeared, where RL algorithms combined with optimization or guidance algorithms have also been useful to produce more intelligent input. However, none of the authors have stated which method they utilized to produce that input. Additionally, tools like AFL or libFuzzer utilize a plethora of mutation methods, which in the case of AFL is utilized along with a GA. Although, the last is considered grey-box, falling in a different category than the one being presented in this thesis.

Regarding test generation, most of the authors relied on fuzzy testing, specification-testing and property-based testing. However, other approaches have also been explored.

RQ3 is mainly addressed in the systematic review presented in Chapter 2.

### 5.3.4 RQ4 – Is Reinforcement Learning an alternative solution to what would be considered a search-based problem?

The work developed and presented in this thesis shows that to implement RL in the context of Software testing is not an easy task, and it becomes especially harder when this process occurs in a black-box scenery, which only makes the context much more stochastic. Search-based problems usually fall under two categories which an exploratory search or an exploitable search should be performed. The work and case study conducted shows that indeed RL can be a good alternative to solve search-based problems. Depending on the domain it may even outperform many other types of algorithms because of its ability to learn on the context in relation to its actions and states.

RQ4 is mainly addressed in systematic review in Chapter 2 and in Chapter 4.

## 5.4 Limitations and Future Work

Despite the success of the work developed and presented, some limitations were recognized and therefore plans for future work exist.

Regarding the solution as a whole, at the moment, it was decided to focus on 5 different datatypes, which seem reasonable. However, when faced with bigger APIs, the range of accepted data types should be broader. Additionally, when faced with functionalities that require specific input values, the tool is prone to fail, because of the mechanism that was implemented for the identification numbers which identify certain entities of these systems. Nevertheless, to achieve better results, other fixed parameter values should be able to be added. Moreover, 9 mutation methods are not enough for the task of producing faulty input values to test a broad range of inputs, even though the methods implemented already allowed for success. A broader range of algorithms should also be available especially to compare and evaluate the performance of each algorithm or even to complement one another if the results vary by much. The tool's explainability regarding vulnerability findings should be much more robust. Currently, the tool relies on the feedback provided by the API under test, to understand and document the flaw, which makes sense considering that the tool is dealing with REST APIs in a black-box scenario. Nonetheless, the results obtained show that even when receiving 2XX HTTP status code in responses, sometimes these contained an error message or were not supposed to have worked considering the scenario. This means that the status codes do not always reflect the success of the operation and as such, the system should consider analysing the response content to verify if a vulnerability was found to be better documented. Additionally, to compliment the report of vulnerabilities, an analysis should be made at the level of the fields which trigger certain vulnerabilities. Even though, the AUTH module is already implemented, because of time constraints, the authors were unable to integrate it with the FuzzCore to enhance the system's security.

Regarding the RL algorithm itself, the work focused on mutating, in every step of the training process, the inputs by their datatype, which for the scope of this thesis can be considered reasonable. Nonetheless, the results demonstrated in this work show that the input fields of each function should also be evaluated along the way, as measure to ensure that faulty inputs are not being mutated and losing its faultiness. Moreover, the consideration of the live coverage as a decision metric during the testing exercise could be valuable for the algorithm to balance exploration with exploitation. In this work to ensure that the algorithm reached some sort of vulnerability exploration decay was implemented, but the coverage metric could be much more interesting and insightful to the algorithm. Moreover, currently the tool creates a separate agent for each function, which the authors still consider the best approach, however, all agents are parameterized the same manner. Therefore, the future work should include a parameter search for each function because the metrics can vary depending on the size and complexity of each function.

# References

[1] R. Torkar, “Towards Automated Software Testing : Techniques, Classifications and Frameworks,” Blekinge Institute of Technology, 2006.

[2] I. Sommerville *et al.*, “Software Engineering (2011 - 9th edition),” 2011.

[3] M. Maleshkova, C. Pedrinaci, and J. Domingue, “Investigating Web APIs on the world wide Web,” *Proceedings - 8th IEEE European Conference on Web Services, ECOWS 2010*, pp. 107–114, 2010, doi: 10.1109/ECOWS.2010.9.

[4] “Gecad - Home.” http://www.gecad.isep.ipp.pt/GECAD/Pages/Presentation/Home.aspx (accessed Apr. 19, 2021).

[5] G. Tassey, “The Economic Impacts of Inadequate Infrastructure for Software Testing Final Report Prepared for,” 2002.

[6] H. Krasner, “The Cost of Poor Software Quality in the US: A 2020 Report,” 2020.

[7] “Software Testing Statistics - TrueList 2022.”

[8] “25+ Cyber Security Vulnerability Statistics and Facts of 2022.”

[9] S. Wannous, T. Dias, E. Maia, I. Praça, and A. R. Faria, “Multiple Domain Security Awareness for Factories of the Future,” 2022, pp. 29–40. doi: 10.1007/978-3-031-18697-4_3.

[10] E. Maia, S. Wannous, T. Dias, I. Praça, and A. Faria, “Holistic Security and Safety for Factories of the Future,” *Sensors*, vol. 22, no. 24, p. 9915, Dec. 2022, doi: 10.3390/s22249915.

[11] J. Vitorino, T. Dias, T. Fonseca, E. Maia, and I. Praça, “Constrained Adversarial Learning and its applicability to Automated Software Testing: a systematic review,” Mar. 2023, Accessed: Sep. 07, 2023. [Online]. Available: https://arxiv.org/abs/2303.07546v1

[12] T. Dias, A. Batista, E. Maia, and I. Praça, “TestLab: An Intelligent Automated Software Testing Framework,” pp. 355–364, 2023, doi: 10.1007/978-3-031-38318-2_35.

[13] D. Moher *et al.*, “Preferred reporting items for systematic review and meta-analysis protocols (PRISMA-P) 2015 statement,” *Revista Espanola de Nutricion Humana y Dietetica*, vol. 20, no. 2, pp. 148–160, Jan. 2016, doi: 10.1186/2046-4053-4-1/TABLES/4.

[14] “ACM Digital Library.” https://dl.acm.org/ (accessed Nov. 07, 2022).

[15] “IEEE Xplore.” https://ieeexplore.ieee.org/Xplore/home.jsp (accessed Nov. 07, 2022).

[16] "2021 Impact Factors - Released." https://www.mdpi.com/about/announcements/4095?utm_campaign=corpnews_if21_announcement&utm_medium=social_corp&utm_source=search&gclid=CjwKCAjw8JKbBhBYEiwAs3sxN3dyZ5IeI5GVCLGHM_33Gz8UOMA7ozHAS06guRWbaD-A7YMJLI5SWhoCnFgQAvD_BwE (accessed Nov. 07, 2022).

[17] "ScienceDirect.com | Science, health and medical journals, full text articles and books." https://www.sciencedirect.com/ (accessed Nov. 07, 2022).

[18] "IEEE Standard for Software Quality Assurance Processes," *IEEE Std 730-2014 (Revision of IEEE Std 730-2002)*, pp. 1–138, 2014, doi: 10.1109/IEEESTD.2014.6835311.

[19] A. Anand and A. Uddin, "Importance of Software Testing in the Process of Software Development," *IJSRD-International Journal for Scientific Research & Development|*, vol. 6, no. February, pp. 2321–0613, 2019.

[20] S. Yang, D. Towey, Z. Q. Zhou, and T. Y. Chen, "Preparing Software Quality Assurance Professionals: Metamorphic Exploration for Machine Learning," in *TALE 2019 - 2019 IEEE International Conference on Engineering, Technology and Education*, Institute of Electrical and Electronics Engineers Inc., Dec. 2019. doi: 10.1109/TALE48000.2019.9225946.

[21] I. Karac and B. Turhan, "What Do We (Really) Know about Test-Driven Development?," *IEEE Softw*, vol. 35, no. 4, pp. 81–85, Jul. 2018, doi: 10.1109/MS.2018.2801554.

[22] K. Petersen and C. Wohlin, "The effect of moving from a plan-driven to an incremental software development approach with agile practices: An industrial case study," *Empir Softw Eng*, vol. 15, no. 6, pp. 654–693, Dec. 2010, doi: 10.1007/s10664-010-9136-6.

[23] D. M. Rafi, K. R. K. Moses, K. Petersen, and M. V Mäntylä, "Benefits and limitations of automated software testing: Systematic literature review and practitioner survey," in *2012 7th International Workshop on Automation of Software Test (AST)*, 2012, pp. 36–42. doi: 10.1109/IWAST.2012.6228988.

[24] A. Martin-Lopez, A. Arcuri, S. Segura, and A. Ruiz-Cortés, "Black-Box and White-Box Test Case Generation for RESTful APIs: Enemies or Allies?," 2021. doi: 10.1109/ISSRE52982.2021.00034.

[25] L. Rajamanickam, N. A. B. Mat Saat, and S. N. Binti Daud, "Software Testing: The Generation Tools," *International Journal of Advanced Trends in Computer Science and Engineering*, vol. 8, no. 2, pp. 231–234, Apr. 2019, doi: 10.30534/ijatcse/2019/20822019.

[26] T. Murnane and K. Reed, "On the Effectiveness of Mutation Analysis as a Black Box Testing Technique.," 2001, pp. 12–20.

[27] S. Nidhra, “Black Box and White Box Testing Techniques - A Literature Review,” *International Journal of Embedded Systems and Applications*, vol. 2, no. 2, pp. 29–50, Jun. 2012, doi: 10.5121/ijesa.2012.2204.

[28] F. Saglietti, N. Oster, and F. Pinte, “White and grey-box verification and validation approaches for safety- and security-critical software systems,” *Information Security Technical Report*, vol. 13, no. 1, pp. 10–16, 2008, doi: 10.1016/j.istr.2008.03.002.

[29] V. Garousi and J. Zhi, “A survey of software testing practices in Canada,” *Journal of Systems and Software*, vol. 86, no. 5, pp. 1354–1376, May 2013, doi: 10.1016/j.jss.2012.12.051.

[30] R. Hamlet, “Random Testing,” in *Encyclopedia of Software Engineering*, Hoboken, NJ, USA: John Wiley & Sons, Inc., 2002. doi: 10.1002/0471028959.sof268.

[31] I. Jovanović, “Software Testing Methods and Techniques”.

[32] V. Garousi and M. V Mäntylä, “A systematic literature review of literature reviews in software testing,” *Inf Softw Technol*, vol. 80, pp. 195–216, 2016, doi: 10.1016/j.infsof.2016.09.002.

[33] P. Bourque, R. Dupuis, A. Abran, J. W. Moore, and L. Tripp, “The Emerging Consensus on the Software Engineering Body of Knowledge”, Accessed: Sep. 04, 2023. [Online]. Available: http://www.swebok.org

[34] A. Qazi, A. Rauf, and N. M. Minhas, “A Systematic Review of Use Cases based Software Testing Techniques,” *International Journal of Software E ngineering and Its Applications*, vol. 10, pp. 337–360, 2016, doi: 10.14257/ijseia.2016.10.11.28.

[35] A. Martin-Lopez, S. Segura, and A. Ruiz-Cortés, “Test Coverage Criteria for RESTful Web APIs,” in *Proceedings of the 10th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*, in A-TEST 2019. New York, NY, USA: Association for Computing Machinery, 2019, pp. 15–21. doi: 10.1145/3340433.3342822.

[36] A. Martin-Lopez, S. Segura, and A. Ruiz-Cortés, “RESTest: Automated Black-Box Testing of RESTful Web APIs,” in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, in ISSTA 2021. New York, NY, USA: Association for Computing Machinery, 2021, pp. 682–685. doi: 10.1145/3460319.3469082.

[37] O. Baniaș, D. Florea, R. Gyalai, and D.-I. Curiac, “Automated Specification-Based Testing of REST APIs,” *Sensors*, vol. 21, no. 16, 2021, doi: 10.3390/s21165375.

[38] V. Atlidakis, P. Godefroid, and M. Polishchuk, “RESTler: Stateful REST API Fuzzing,” in *ICSE 2019*, 2019.

[39] H. Ed-Douibi, J. L. C. Izquierdo, and J. Cabot, "Automatic Generation of Test Cases for REST APIs: A Specification-Based Approach," *2018 IEEE 22nd International Enterprise Distributed Object Computing Conference (EDOC)*, pp. 181–190, 2018.

[40] S. Karlsson, A. Causevic, and D. Sundmark, "QuickREST: Property-based Test Generation of OpenAPI-Described RESTful APIs." arXiv, 2019. doi: 10.48550/ARXIV.1912.09686.

[41] S. O. Haraldsson, J. R. Woodward, and A. I. E. Brownlee, "The Use of Automatic Test Data Generation for Genetic Improvement in a Live System," in *Proceedings of the 10th International Workshop on Search-Based Software Testing*, in SBST '17. IEEE Press, 2017, pp. 28–31.

[42] Y. Liu *et al.*, "Morest: Model-Based RESTful API Testing with Execution Feedback," in *Proceedings of the 44th International Conference on Software Engineering*, in ICSE '22. New York, NY, USA: Association for Computing Machinery, 2022, pp. 1406–1417. doi: 10.1145/3510003.3510133.

[43] M. Biagiola, A. Stocco, F. Ricca, and P. Tonella, "Diversity-Based Web Test Generation," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, in ESEC/FSE 2019. New York, NY, USA: Association for Computing Machinery, 2019, pp. 142–153. doi: 10.1145/3338906.3338970.

[44] A. Martin-Lopez, "AI-Driven Web API Testing," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings*, in ICSE '20. New York, NY, USA: Association for Computing Machinery, 2020, pp. 202–205. doi: 10.1145/3377812.3381388.

[45] S. Reddy, C. Lemieux, R. Padhye, and K. Sen, "Quickly Generating Diverse Valid Test Inputs with Reinforcement Learning," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, in ICSE '20. New York, NY, USA: Association for Computing Machinery, 2020, pp. 1410–1421. doi: 10.1145/3377811.3380399.

[46] Y. Zheng *et al.*, "Automatic Web Testing Using Curiosity-Driven Reinforcement Learning," in *Proceedings of the 43rd International Conference on Software Engineering*, in ICSE '21. IEEE Press, 2021, pp. 423–435. doi: 10.1109/ICSE43902.2021.00048.

[47] "Home - OpenAPI Initiative."

[48] Q. Shen, M. Wen, L. Zhang, L. Wang, L. Shen, and J. Cheng, "A systematic review of fuzzy testing for information systems and applications," in *2021 2nd International Conference on Electronics, Communications and Information Technology (CECIT)*, IEEE, Dec. 2021, pp. 156–162. doi: 10.1109/CECIT53797.2021.00035.

[49] M. Böhme and J. Metzman, "On the Reliability of Coverage-Based Fuzzer Benchmarking," 2022, doi: 10.1145/3510003.3510230.

[50] A. Hazimeh, S. Adrian Herrera, and A. Mathias Payer, “Magma: A Ground-Truth Fuzzing Benchmark,” *In Proc. ACM Meas. Anal. Comput. Syst*, vol. 4, 2020, doi: 10.1145/3428334.

[51] A. Fioraldi, A. Mantovani, D. Maier, and D. Balzarotti, “Dissecting American Fuzzy Lop – A FuzzBench Evaluation.” Jan. 2022. doi: 10.13140/RG.2.2.13803.82722.

[52] G. J. Saavedra, K. N. Rodhouse, D. M. Dunlavy, and P. W. Kegelmeyer, “A Review of Machine Learning Applications in Fuzzing,” Jun. 2019.

[53] “Peach Fuzzer.” https://peachtech.gitlab.io/peach-fuzzer-community/ (accessed Jan. 05, 2023).

[54] “Peach Tech / peach-fuzzer-community · GitLab.” https://gitlab.com/peachtech/peach-fuzzer-community (accessed Jan. 10, 2023).

[55] “boofuzz: Network Protocol Fuzzing for Humans — boofuzz 0.4.1 documentation.” https://boofuzz.readthedocs.io/en/stable/ (accessed Jan. 05, 2023).

[56] “OpenRCE/sulley: A pure-python fully automated and unattended fuzzing framework.” https://github.com/OpenRCE/sulley (accessed Jan. 11, 2023).

[57] “google/honggfuzz: Security oriented software fuzzer. Supports evolutionary, feedback-driven fuzzing based on code coverage (SW and HW based).” https://github.com/google/honggfuzz (accessed Jan. 05, 2023).

[58] “american fuzzy lop.” https://lcamtuf.coredump.cx/afl/ (accessed Jan. 05, 2023).

[59] “libFuzzer – a library for coverage-guided fuzz testing. — LLVM 16.0.0git documentation.” https://llvm.org/docs/LibFuzzer.html (accessed Jan. 05, 2023).

[60] A. Pramanik and A. Tayade, “Study and Comparison of General Purpose Fuzzers,” 2019. Accessed: Jan. 11, 2023. [Online]. Available: https://theultramarine19.github.io/data/736.pdf

[61] W. Drozd and M. D. Wagner, “FuzzerGym: A Competitive Framework for Fuzzing and Learning,” Jul. 2018.

[62] J. Mccarthy, “What is Artificial Intelligence?,” 2004.

[63] J. A. Bullinaria, “IAI : The Roots, Goals and Sub-fields of AI”.

[64] T. Sejnowski, “The Deep Learning Revolution (The MIT Press),” 2018.

[65] A. L. Samuel, “Some Studies in Machine Learning Using the Game of Checkers,” *IBM J Res Dev*, vol. 3, no. 3, pp. 210–229, Jul. 1959, doi: 10.1147/rd.33.0210.

[66] Y. Sibaroni, D. H. Widyantoro, and M. L. Khodra, “Extend relation identification in scientific papers based on supervised machine learning,” in *2016 International*

*Conference on Advanced Computer Science and Information Systems (ICACSIS)*, IEEE, Oct. 2016, pp. 379–384. doi: 10.1109/ICACSIS.2016.7872724.

[67] F. A. Breve and D. C. G. Pedronette, "Combined unsupervised and semi-supervised learning for data classification," in *2016 IEEE 26th International Workshop on Machine Learning for Signal Processing (MLSP)*, IEEE, Sep. 2016, pp. 1–6. doi: 10.1109/MLSP.2016.7738877.

[68] P. Shen, X. Du, and C. Li, "Distributed Semi-Supervised Metric Learning," *IEEE Access*, vol. 4, pp. 8558–8571, 2016, doi: 10.1109/ACCESS.2016.2632158.

[69] N. Shimkin, "Learning in Complex Systems: Reinforcement Learning Basic Algorithms," 2009.

[70] R. S. Sutton and A. G. Barto, "Reinforcement Learning: An Introduction," *IEEE Trans Neural Netw*, vol. 9, no. 5, pp. 1054–1054, Sep. 1998, doi: 10.1109/TNN.1998.712192.

[71] D. Silver *et al.*, "Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm," Dec. 2017.

[72] F. Kominis and H. Geffner, "Beliefs In Multiagent Planning: From One Agent to Many," *Proceedings of the International Conference on Automated Planning and Scheduling*, vol. 25, pp. 147–155, Apr. 2015, doi: 10.1609/icaps.v25i1.13726.

[73] M. L. Puterman, "Chapter 8 Markov decision processes," 1990, pp. 331–434. doi: 10.1016/S0927-0507(05)80172-0.

[74] S. Peng, "Stochastic Hamilton–Jacobi–Bellman Equations," *SIAM J Control Optim*, vol. 30, no. 2, pp. 284–304, Mar. 1992, doi: 10.1137/0330018.

[75] H. Zhang and T. Yu, "Taxonomy of Reinforcement Learning Algorithms," in *Deep Reinforcement Learning*, Singapore: Springer Singapore, 2020, pp. 125–133. doi: 10.1007/978-981-15-4095-0_3.

[76] V. Mnih *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, Feb. 2015, doi: 10.1038/nature14236.

[77] C. J. C. H. Watkins and P. Dayan, "Q-learning," *Mach Learn*, vol. 8, no. 3–4, pp. 279–292, May 1992, doi: 10.1007/BF00992698.

[78] K. Gurney, *An Introduction to Neural Networks*. CRC Press, 2018. doi: 10.1201/9781315273570.

[79] V. Mnih *et al.*, "Playing Atari with Deep Reinforcement Learning," Dec. 2013.

[80] S. Lang, F. Behrendt, N. Lanzerath, T. Reggelin, and M. Muller, "Integration of Deep Reinforcement Learning and Discrete-Event Simulation for Real-Time Scheduling of a

Flexible Job Shop Production," in *2020 Winter Simulation Conference (WSC)*, IEEE, Dec. 2020, pp. 3057–3068. doi: 10.1109/WSC48552.2020.9383997.

[81] P. Munro *et al.*, "Bellman Equation," in *Encyclopedia of Machine Learning*, Boston, MA: Springer US, 2011, pp. 97–97. doi: 10.1007/978-0-387-30164-8_71.

[82] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal Policy Optimization Algorithms," Jul. 2017.

[83] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal Policy Optimization Algorithms," Jul. 2017.

[84] J. D. Mcgregor and D. A. Sykes, *A Practical Guide to Testing Object-Oriented Software*, 1st ed. jAddison-Wesley Professional, 2001.

[85] P. Godefroid, "Random testing for security," in *Proceedings of the 2nd international workshop on Random testing: co-located with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007)*, New York, NY, USA: ACM, Nov. 2007, pp. 1–1. doi: 10.1145/1292414.1292416.

[86] W. Liang *et al.*, "Advances, challenges and opportunities in creating data for trustworthy AI," *Nat Mach Intell*, vol. 4, no. 8, pp. 669–677, Aug. 2022, doi: 10.1038/s42256-022-00516-1.

[87] G. MISURACA and N. C. VAN, "AI Watch - Artificial Intelligence in public services," *Publications Office of the European Union*, no. July, pp. 1–96, 2020, doi: 10.2760/039619.

[88] K. Charmaz and L. L. Belgrave, "Thinking About Data With Grounded Theory," *Qualitative Inquiry*, vol. 25, no. 8, pp. 743–753, Oct. 2019, doi: 10.1177/1077800418809455.

[89] F. Armour and G. Miller, *Advanced use case modeling: software systems*. Pearson Education, 2000.

[90] R. C. Martin, "Design Principles and Design Patterns," 2000.

[91] C. Larman, *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process (2nd Edition)*. 2001.

[92] S. Brown, "The C4 Model for Software Architecture." https://www.infoq.com/articles/C4-architecture-model/ (accessed May 07, 2021).

[93] P. Kruchten, "Architectural Blueprints-The '4+1' View Model of Software Architecture," 1995.

[94] "Robert C. Martin - Wikipedia." https://en.wikipedia.org/wiki/Robert_C._Martin (accessed May 06, 2021).

[95] “A Solid Guide to SOLID Principles | Baeldung.” https://www.baeldung.com/solid-principles (accessed May 06, 2021).

[96] “The S.O.L.I.D Principles in Pictures | by Ugonna Thelma | Backticks & Tildes | Medium.” https://medium.com/backticks-tildes/the-s-o-l-i-d-principles-in-pictures-b34ce2f1e898 (accessed May 06, 2021).

[97] “GRASP - General Responsibility Assignment Software Patterns Explained - Kamil Grzybek.” http://www.kamilgrzybek.com/design/grasp-explained/ (accessed May 07, 2021).

[98] “Kruchten’s 4 + 1 views of Software Design | by Puneet Sapra | The Mighty Programmer | Medium.” https://medium.com/the-mighty-programmer/kruchtens-views-of-software-design-e9088398c592 (accessed May 07, 2021).

[99] “The C4 model for visualising software architecture.” https://c4model.com/ (accessed May 07, 2021).

[100] “Java | Oracle.” https://www.java.com/pt-BR/ (accessed Apr. 11, 2021).

[101] “Spring | Home.” https://spring.io/ (accessed Jun. 25, 2021).

[102] “MySQL.” https://www.mysql.com/ (accessed Jan. 22, 2023).

[103] I. Dominte, “Introducing Web API,” *Web API Development for the Absolute Beginner*, pp. 3–18, 2023, doi: 10.1007/978-1-4842-9348-5_1.

[104] “Welcome to Python.org.” https://www.python.org/ (accessed Aug. 19, 2021).

[105] “Gym Documentation.” https://www.gymlibrary.dev/ (accessed Jan. 21, 2023).

[106] “NumPy.” https://numpy.org/ (accessed May 04, 2021).

[107] “TextBlob: Simplified Text Processing — TextBlob 0.16.0 documentation.” https://textblob.readthedocs.io/en/dev/ (accessed Aug. 28, 2023).

[108] “requests · PyPI.” https://pypi.org/project/requests/ (accessed Jan. 21, 2023).

[109] “Swagger UI.” https://petstore.swagger.io/ (accessed Jan. 21, 2023).

[110] “lcamtuf’s old blog: Binary fuzzing strategies: what works, what doesn’t.” https://lcamtuf.blogspot.com/2014/08/binary-fuzzing-strategies-what-works.html (accessed Jan. 21, 2023).

[111] “HTTP response status codes - HTTP | MDN.” https://developer.mozilla.org/en-US/docs/Web/HTTP/Status (accessed Jan. 21, 2023).

[112] N. Kantasewi, S. Marukatat, S. Thainimit, and O. Manabu, “Multi Q-Table Q-Learning,” *10th International Conference on Information and Communication Technology for*

*Embedded Systems, IC-ICTES 2019 - Proceedings*, Apr. 2019, doi: 10.1109/ICTEMSYS.2019.8695963.

[113] H. Wang, T. Zariphopoulou, and X. Yu Zhou, “Exploration versus exploitation in reinforcement learning: a stochastic control approach *,” 2019.

[114] B. C. Stadie, S. Levine, and P. Abbeel, “Incentivizing Exploration In Reinforcement Learning With Deep Predictive Models”.

[115] Y. Liu, B. Cao, and H. Li, “Improving ant colony optimization algorithm with epsilon greedy and Levy flight,” *Complex & Intelligent Systems*, vol. 7, no. 4, pp. 1711–1722, Aug. 2021, doi: 10.1007/s40747-020-00138-3.

[116] A. Becue *et al.*, “CyberFactory#1 — Securing the industry 4.0 with cyber-ranges and digital twins,” in *2018 14th IEEE International Workshop on Factory Communication Systems (WFCS)*, IEEE, Jun. 2018, pp. 1–4. doi: 10.1109/WFCS.2018.8402377.

[117] “Home - CyberFactory#1.” https://www.cyberfactory-1.org/home/ (accessed Jan. 21, 2023).

[118] “Maven – Welcome to Apache Maven.” https://maven.apache.org/ (accessed Aug. 29, 2023).

[119] “Docker: Accelerated Container Application Development.” https://www.docker.com/ (accessed Aug. 29, 2023).

[120] “EclEmma - Java Code Coverage for Eclipse.” https://www.jacoco.org/ (accessed Aug. 29, 2023).

[121] D. Corradini, A. Zampieri, M. Pasqua, and M. Ceccato, “Restats: A Test Coverage Tool for RESTful APIs,” *Proceedings - 2021 IEEE International Conference on Software Maintenance and Evolution, ICSME 2021*, pp. 594–598, 2021, doi: 10.1109/ICSME52107.2021.00063.

[122] R. Mahmood, J. Pennington, D. Tsang, T. Tran, and A. Bogle, “A Framework for Automated API Fuzzing at Enterprise Scale,” *Proceedings - 2022 IEEE 15th International Conference on Software Testing, Verification and Validation, ICST 2022*, pp. 377–388, 2022, doi: 10.1109/ICST53961.2022.00018.

[123] D. Corradini, A. Zampieri, M. Pasqua, E. Viglianisi, M. Dallago, and M. Ceccato, “Automated black-box testing of nominal and error scenarios in RESTful APIs,” *Software Testing, Verification and Reliability*, vol. 32, no. 5, p. e1808, Aug. 2022, doi: 10.1002/STVR.1808.

[124] “Extensible Markup Language (XML) 1.1 (Second Edition).” https://www.w3.org/TR/xml11/#sec-white-space (accessed Sep. 01, 2023).

[125] E. Viglianisi, M. Dallago, and M. Ceccato, “RESTTESTGEN: Automated Black-Box Testing of RESTful APIs,” in *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, IEEE, Oct. 2020, pp. 142–152. doi: 10.1109/ICST46399.2020.00024.

[126] A. Tokos, “Evaluating fuzzing tools for automated testing of REST APIs using OpenAPI specification,” 2023, Accessed: Sep. 07, 2023. [Online]. Available: https://www.doria.fi/handle/10024/187395

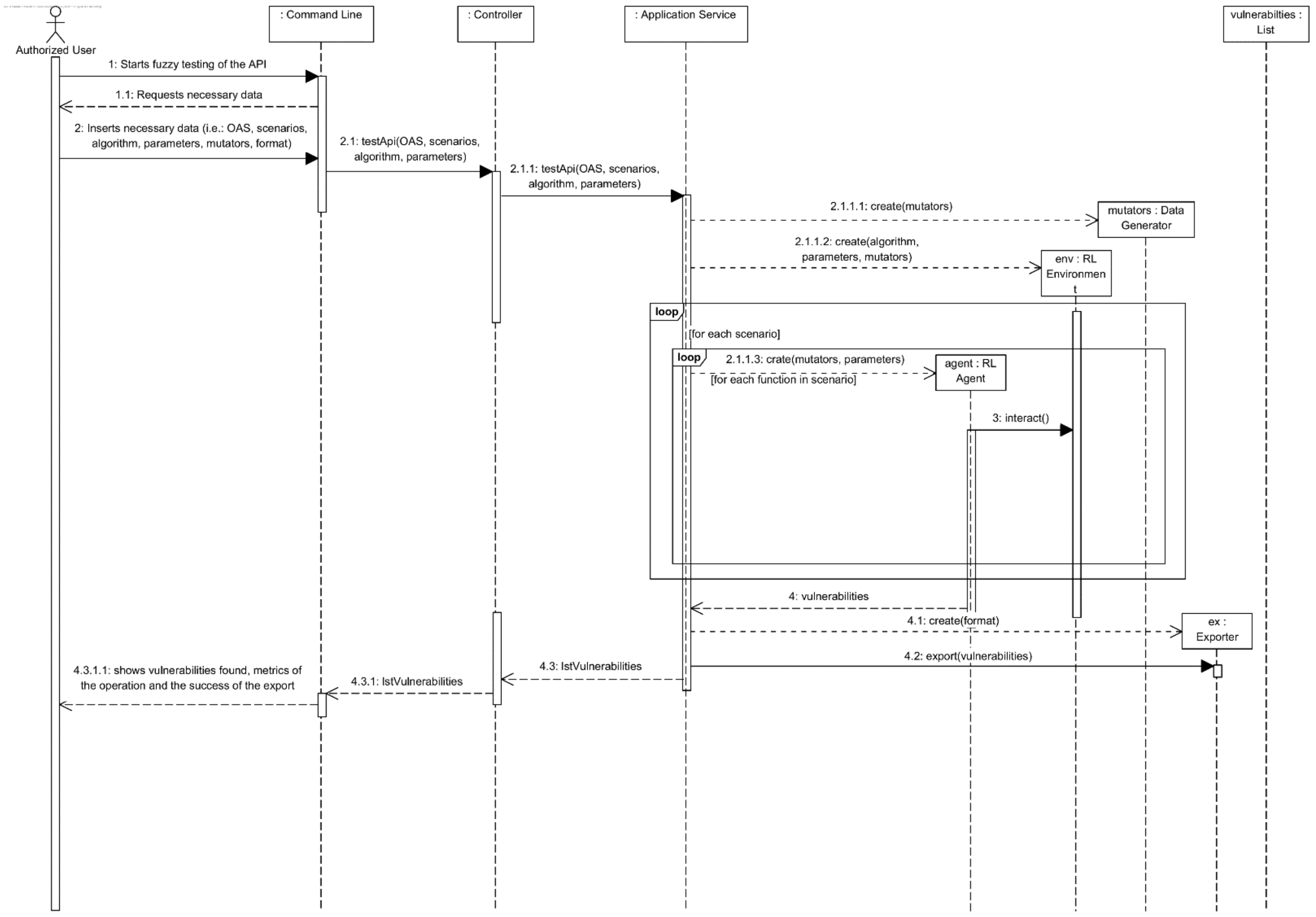
[127] T. Dias, N. Oliveira, N. Sousa, I. Praça, and O. Sousa, “A Hybrid Approach for an Interpretable and Explainable Intrusion Detection System,” *Lecture Notes in Networks and Systems*, vol. 418 LNNS, pp. 1035–1045, 2022, doi: 10.1007/978-3-030-96308-8_96/COVER.

[128] S. Gu, F. Wang, N. P. Patel, J. A. Bourgeois, and J. H. Huang, “A Model for Basic Emotions Using Observations of Behavior in Drosophila,” *Front Psychol*, vol. 10, Apr. 2019, doi: 10.3389/fpsyg.2019.00781.

[129] “Apache Kafka.” https://kafka.apache.org/ (accessed May 05, 2021).

# Appendixes

# Appendix A – Process View



Authorized User
: Command Line
: Controller
: Application Service
vulnerabilties : List
1: Starts fuzzy testing of the API
1.1: Requests necessary data
2: Inserts necessary data (i.e.: OAS, scenarios, algorithm, parameters, mutators, format)
2.1: testApi(OAS, scenarios, algorithm, parameters)
2.1.1: testApi(OAS, scenarios, algorithm, parameters)
2.1.1.1: create(mutators)
mutators : Data Generator
2.1.1.2: create(algorithm, parameters, mutators)
env : RL Environment
loop
[for each scenario]
loop
2.1.1.3: crate(mutators, parameters)
[for each function in scenario]
agent : RL Agent
3: interact()
4: vulnerabilities
4.1: create(format)
ex : Exporter
4.2: export(vulnerabilities)
4.3: lstVulnerabilities
4.3.1: lstVulnerabilities
4.3.1.1: shows vulnerabilities found, metrics of the operation and the success of the export

Figure 26 – Level 3 Process View

## Appendix B- Agent Training Process View

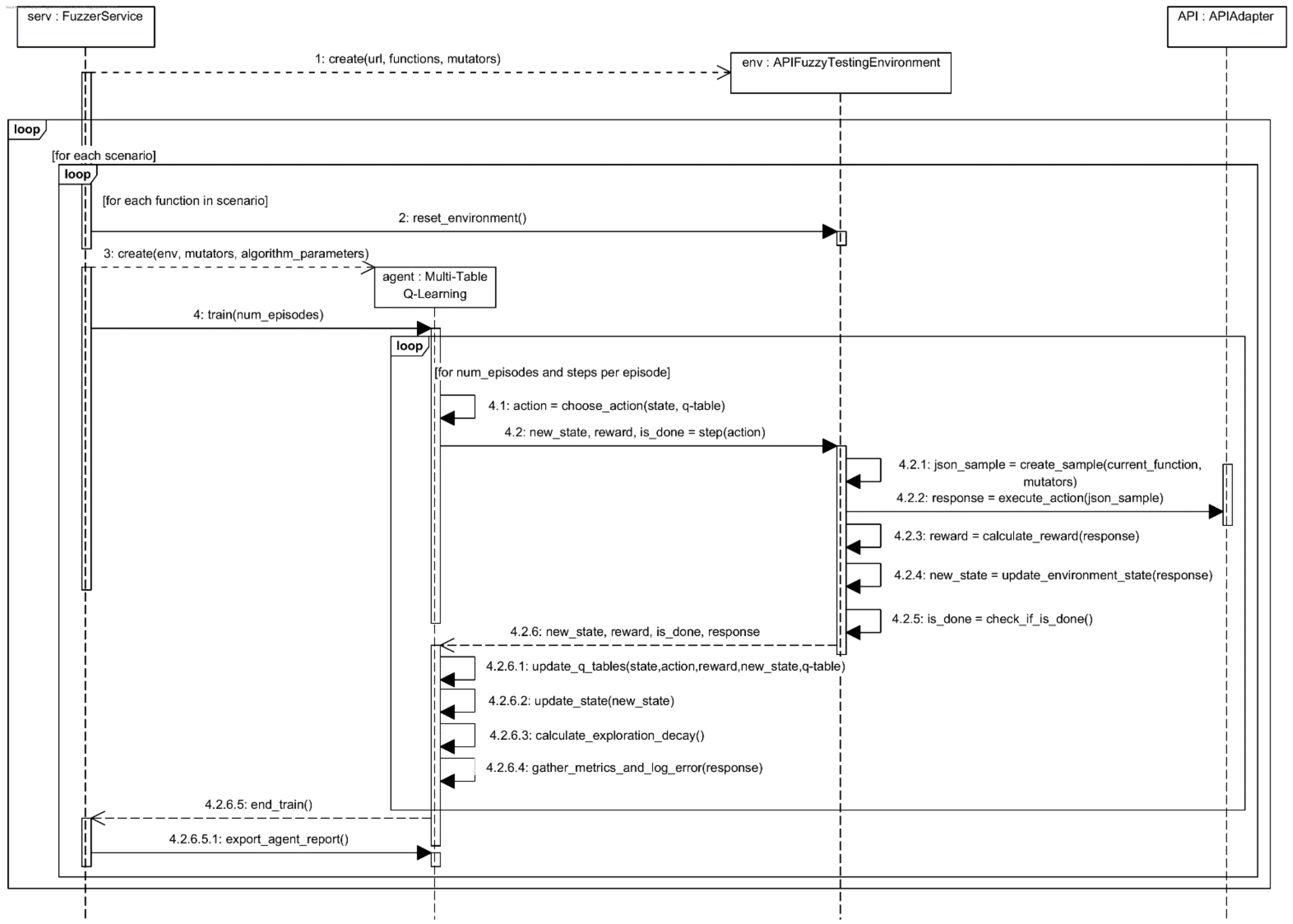

serv : FuzzerService
API : APIAdapter
1: create(url, functions, mutators)
env : APIFuzzyTestingEnvironment
loop
[for each scenario]
loop
[for each function in scenario]
2: reset_environment()
3: create(env, mutators, algorithm_parameters)
agent : Multi-Table Q-Learning
4: train(num_episodes)
loop
[for num_episodes and steps per episode]
4.1: action = choose_action(state, q-table)
4.2: new_state, reward, is_done = step(action)
4.2.1: json_sample = create_sample(current_function, mutators)
4.2.2: response = execute_action(json_sample)
4.2.3: reward = calculate_reward(response)
4.2.4: new_state = update_environment_state(response)
4.2.5: is_done = check_if_is_done()
4.2.6: new_state, reward, is_done, response
4.2.6.1: update_q_tables(state,action,reward,new_state,q-table)
4.2.6.2: update_state(new_state)
4.2.6.3: calculate_exploration_decay()
4.2.6.4: gather_metrics_and_log_error(response)
4.2.6.5: end_train()
4.2.6.5.1: export_agent_report()

Figure 27 – Agent Training Process View

# Appendix C – Petstore API Result Plots

In this appendix the reader can find for each function, the action distribution plots for each datatype considered in the solution (integer, float, boolean, byte and string), the total number of state visits during the agent's training, and the agent's Q-value convergence.

### C.1 – AddPet

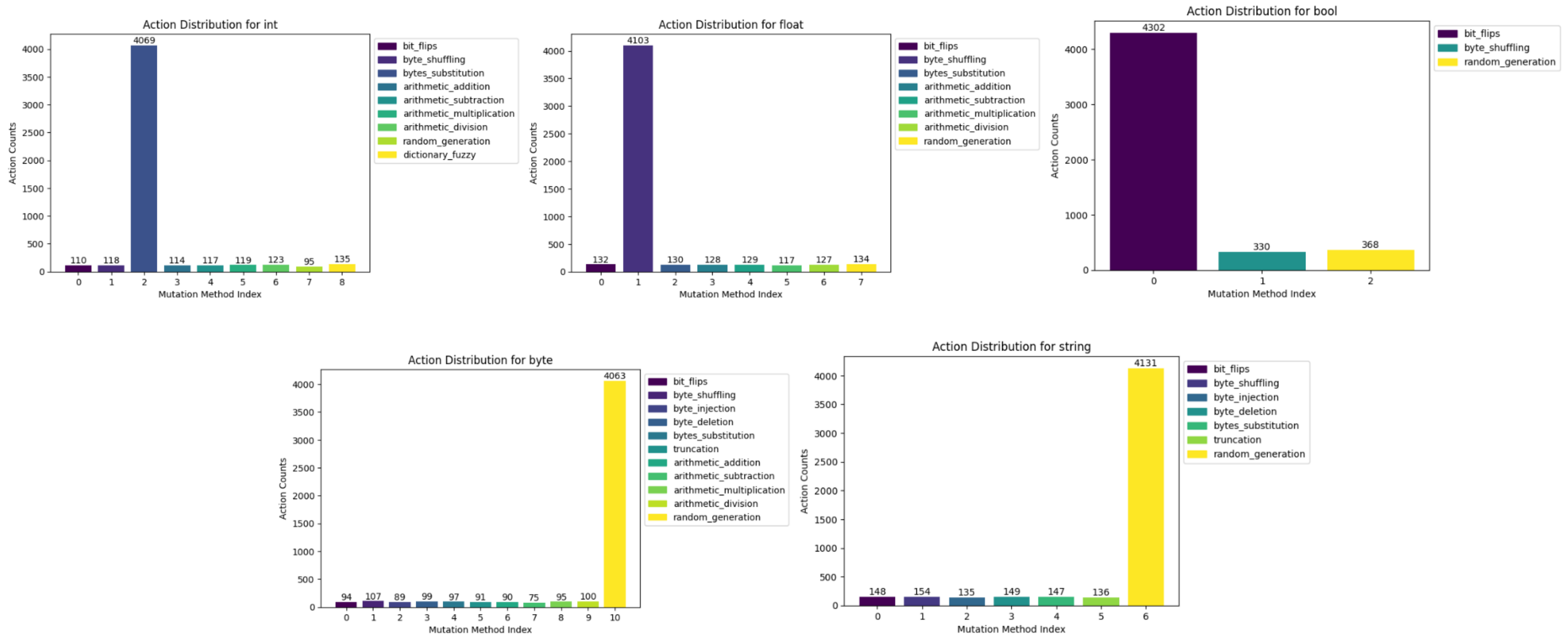


Figure 28 – AddPet Agent's Action Distribution

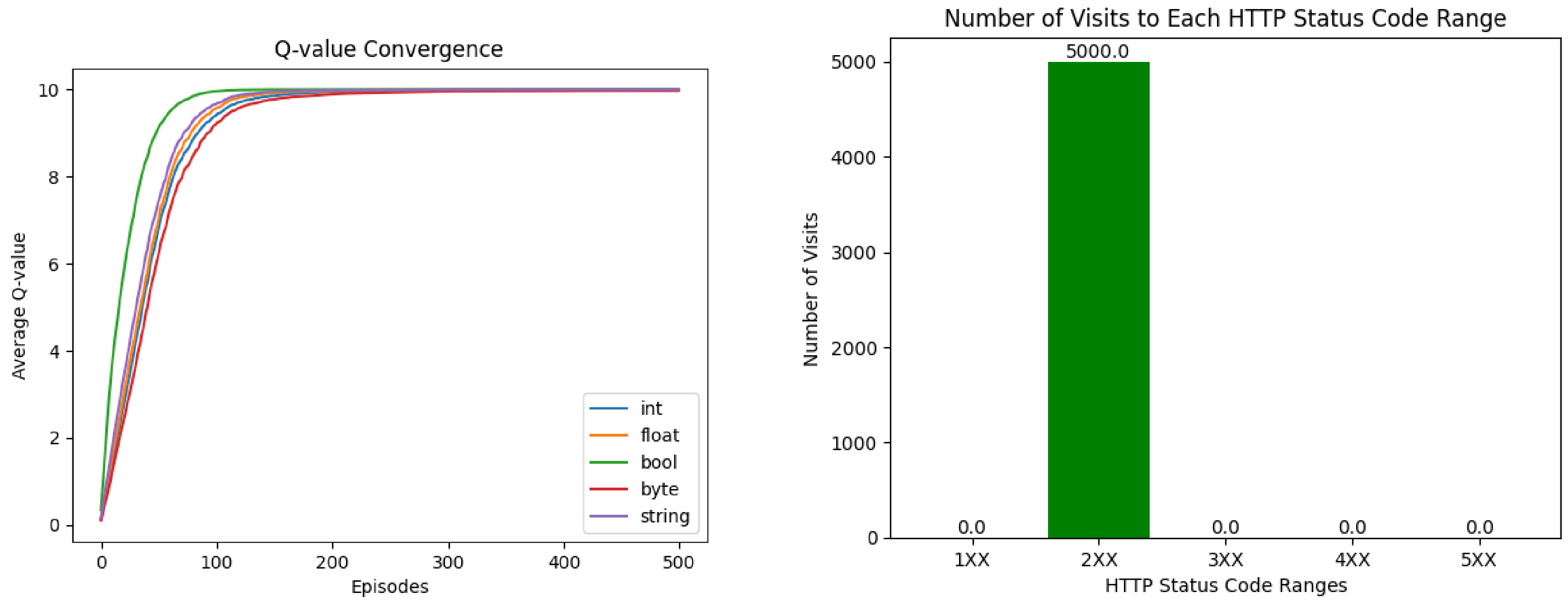


Figure 29 - AddPet Agent’s Q-Value Convergence and State Visits

## C.2 – UpdatePet

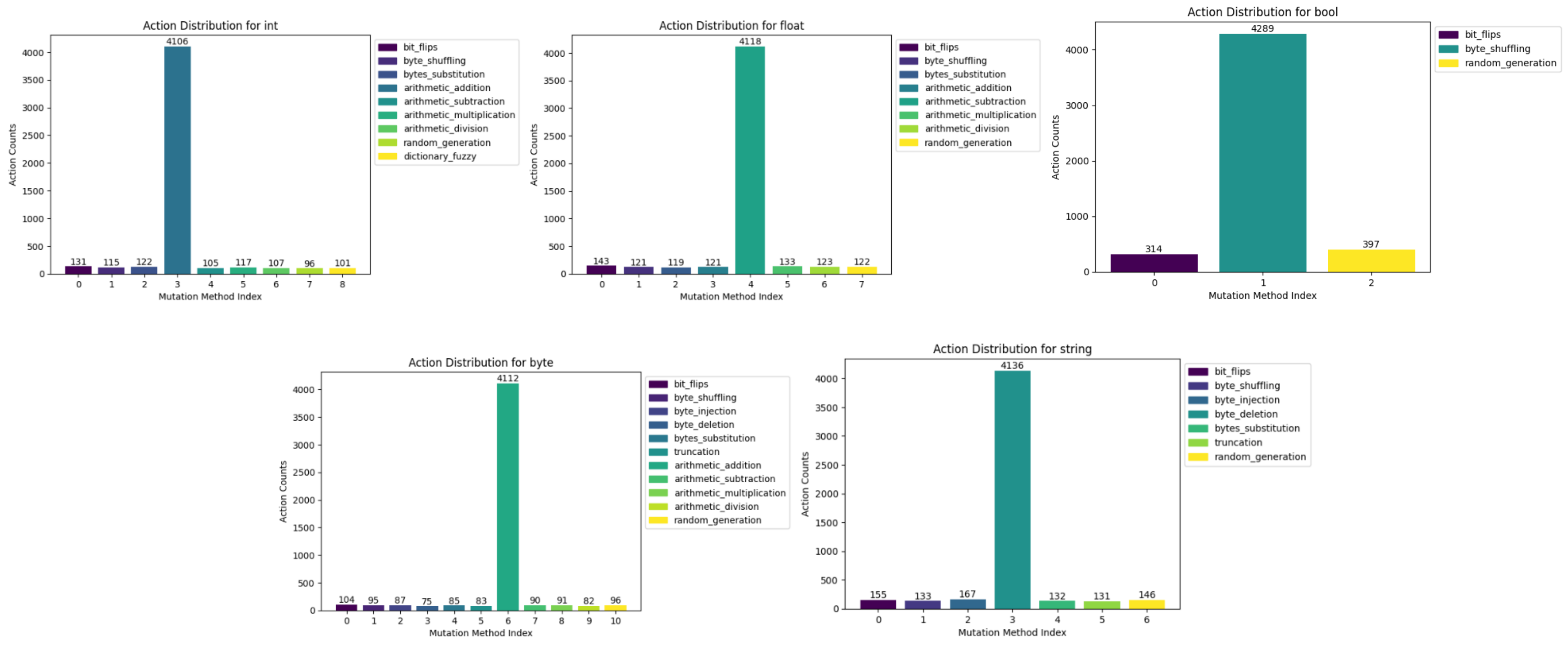
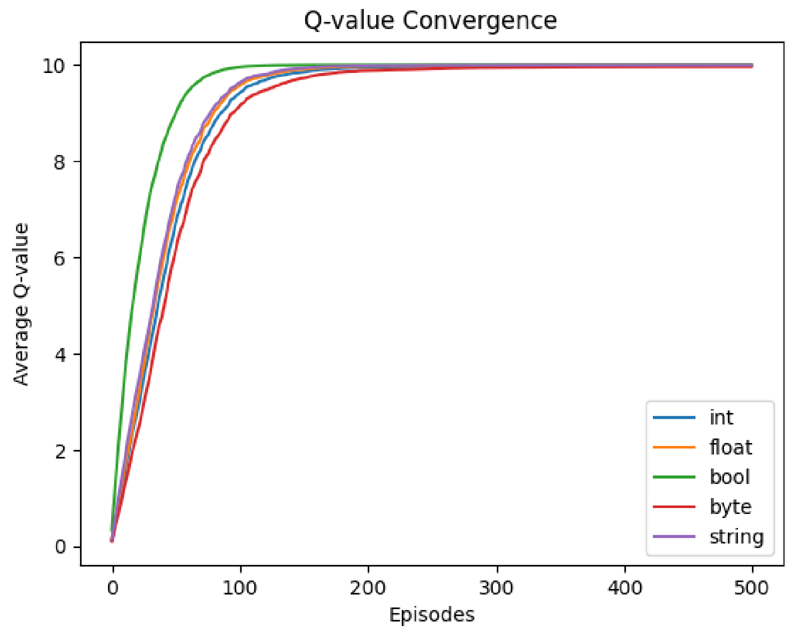


Figure 30 - UpdatePet Agent's Action Distribution



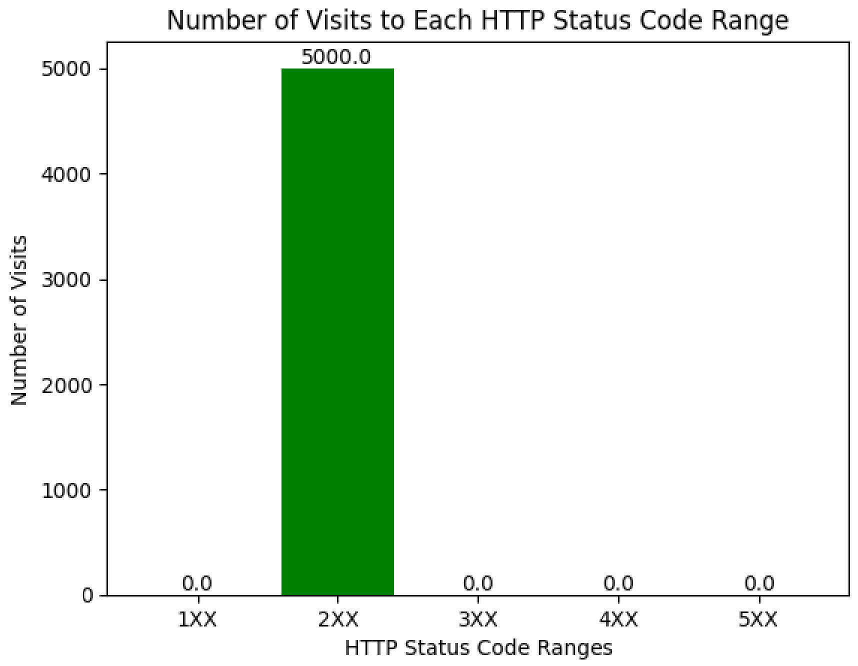


Figure 31 - UpdatePet Agent's Q-Value Convergence and State Visits

## C.3 – GetPetById

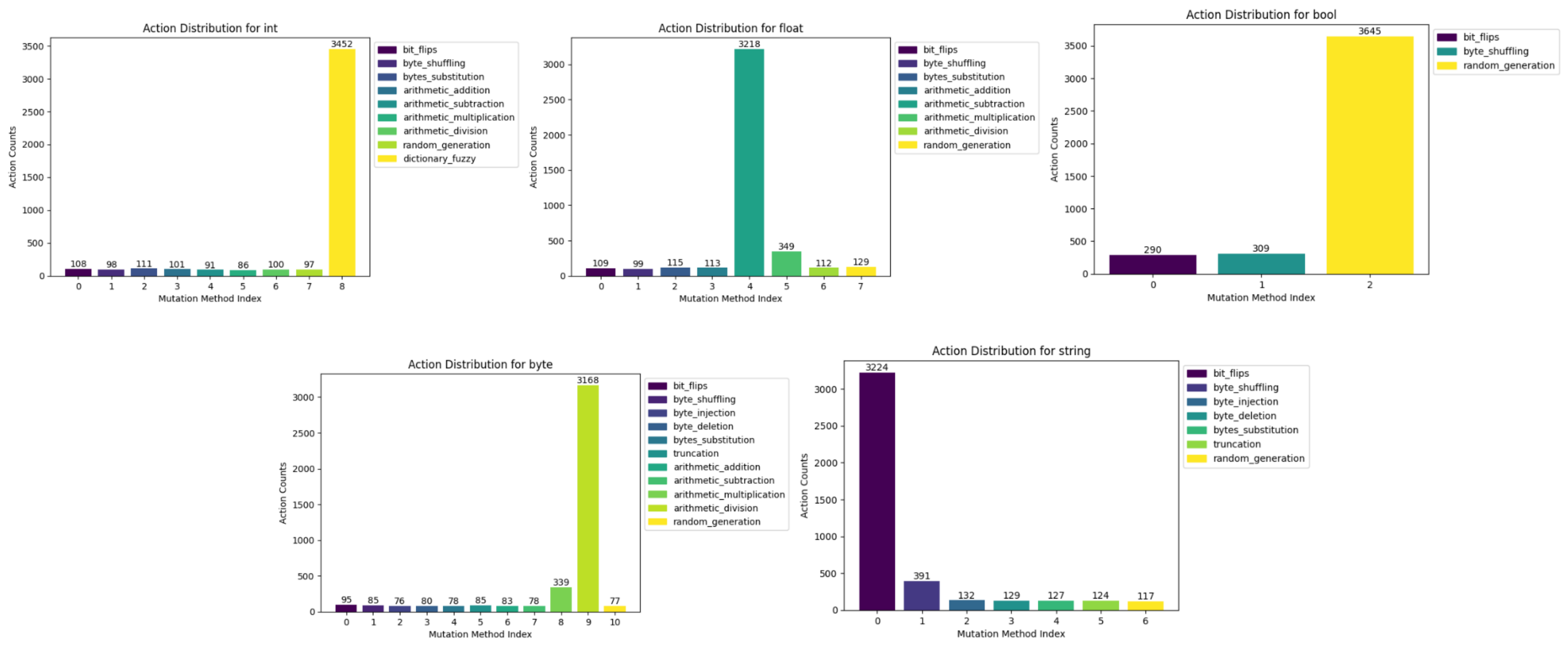
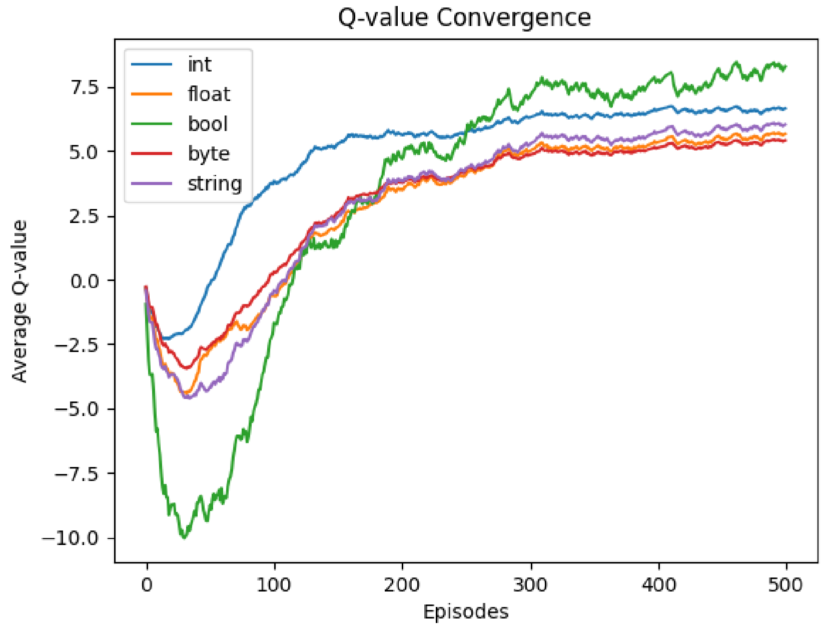


Figure 32 - GetPetById Agent’s Action Distribution



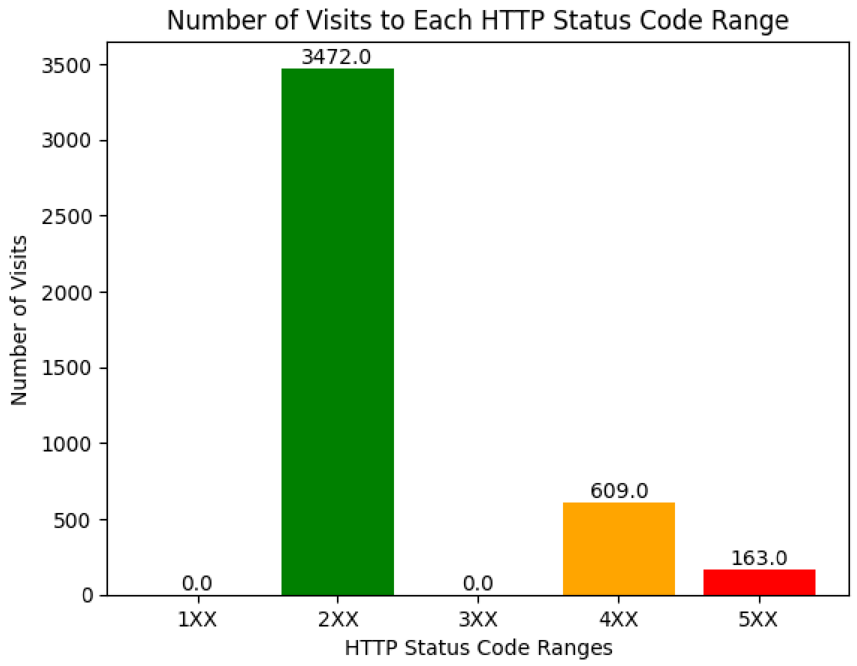


Figure 33 - GetPetById Agent’s Q-Value Convergence and State Visits

## C.4 – FindPetsByStatus

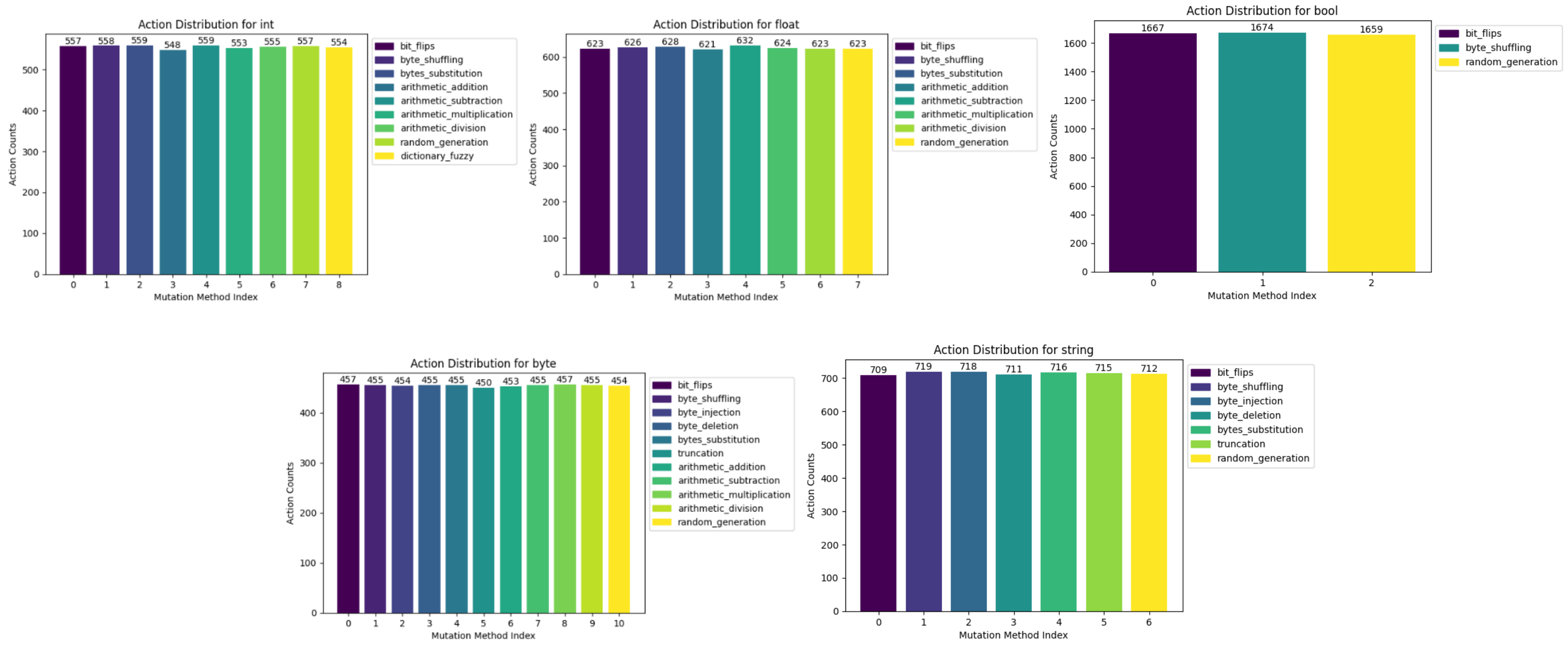
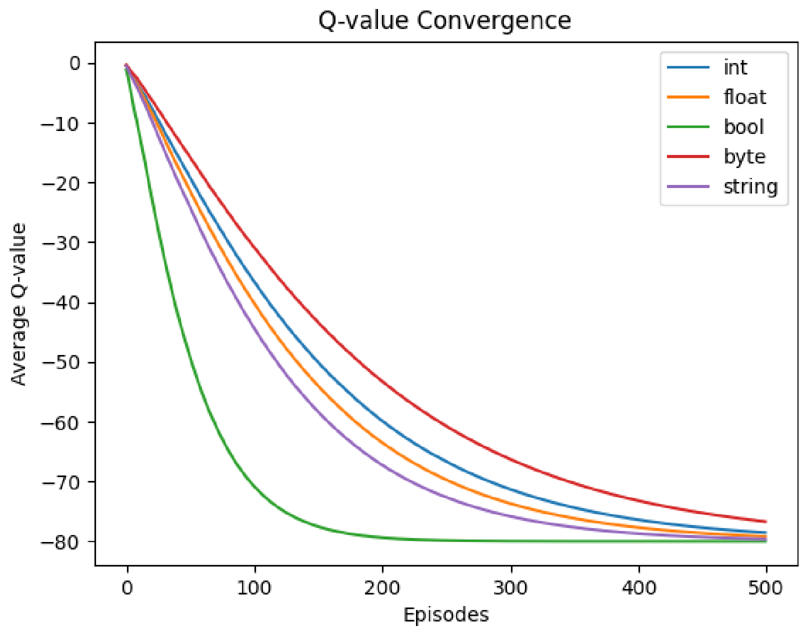


Figure 34 - FindPetsByStatus Agent’s Action Distribution



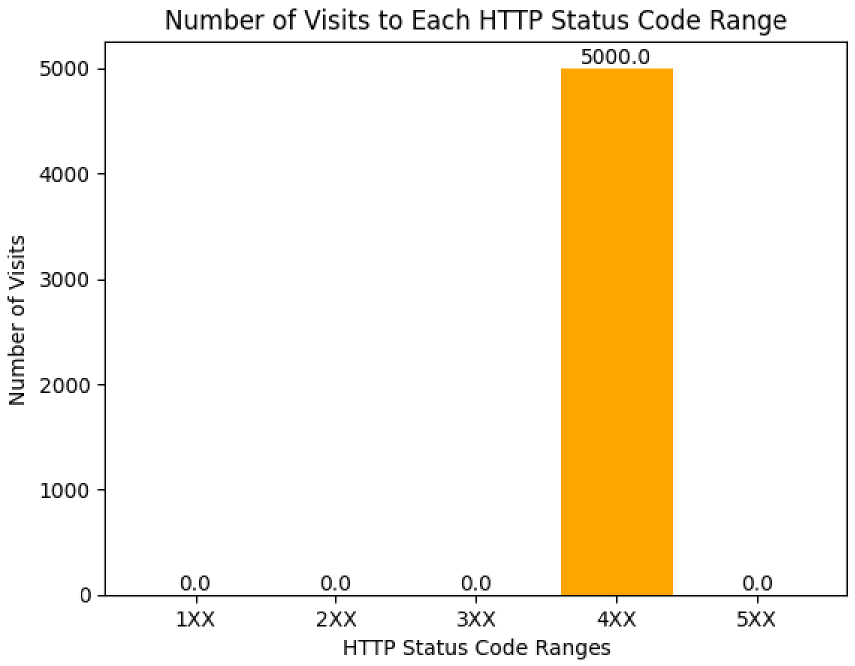


Figure 35 - FindPetsByStatus Agent’s Q-Value Convergence and State Visits

## C.5 – FindPetsByTags

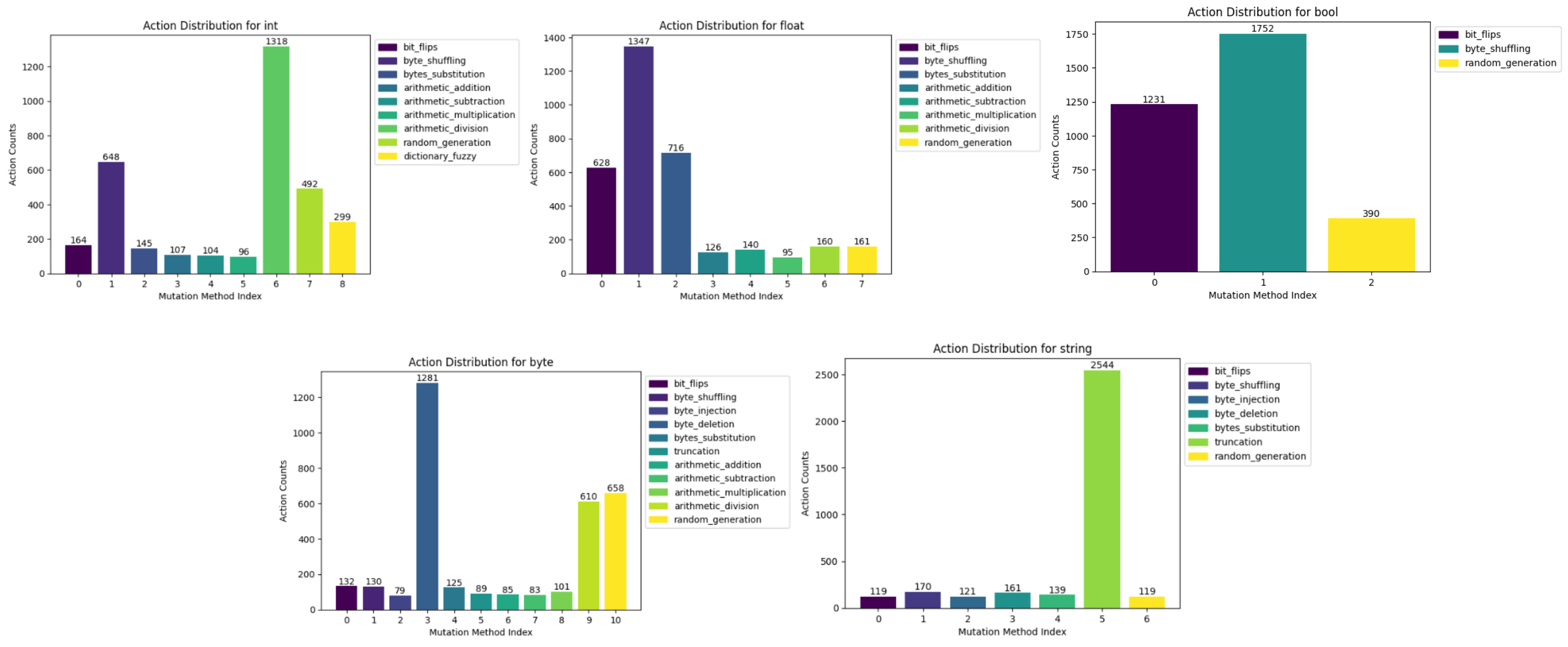
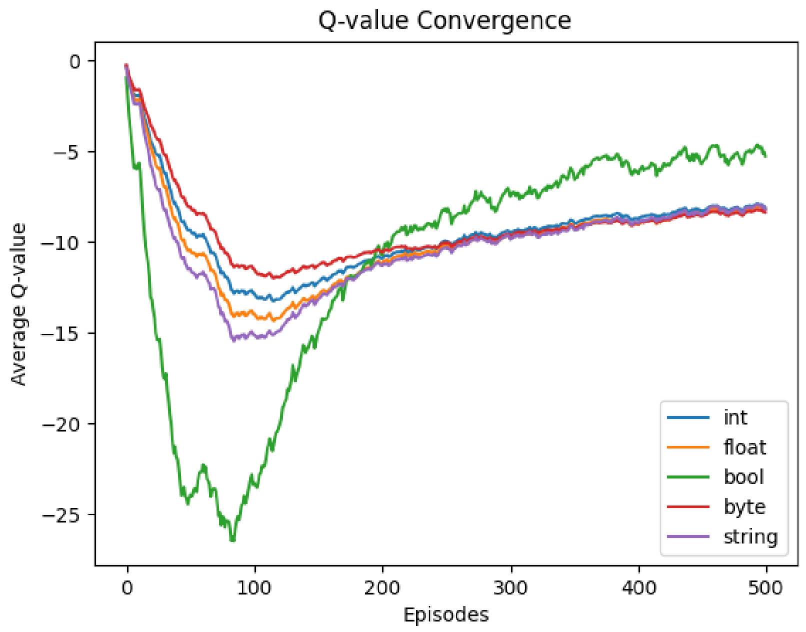


Figure 36 - FindPetsByTags Agent's Action Distribution



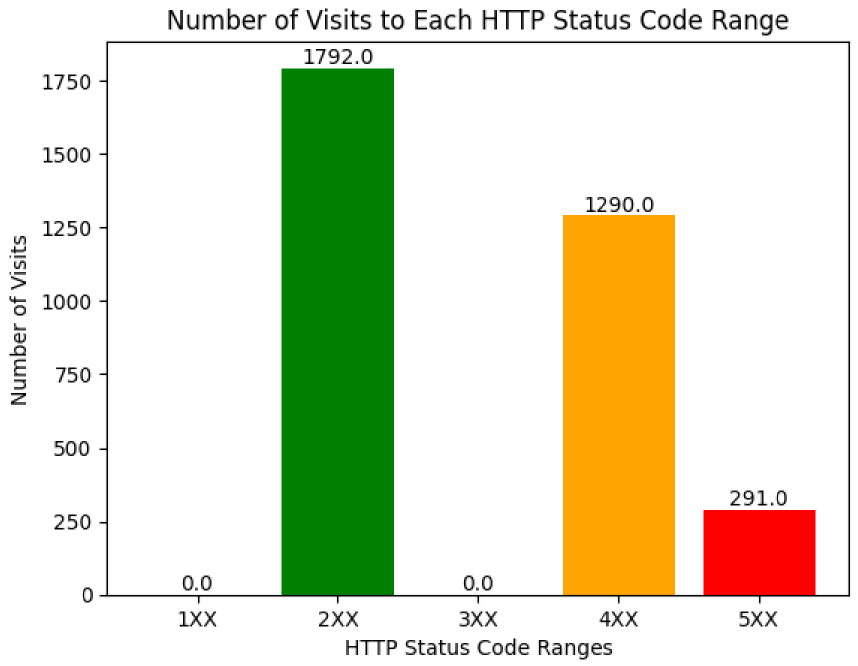


Figure 37 – FindPetsByTags Agent’s Q-Value Convergence and State Visits

## C.6 – UploadFile

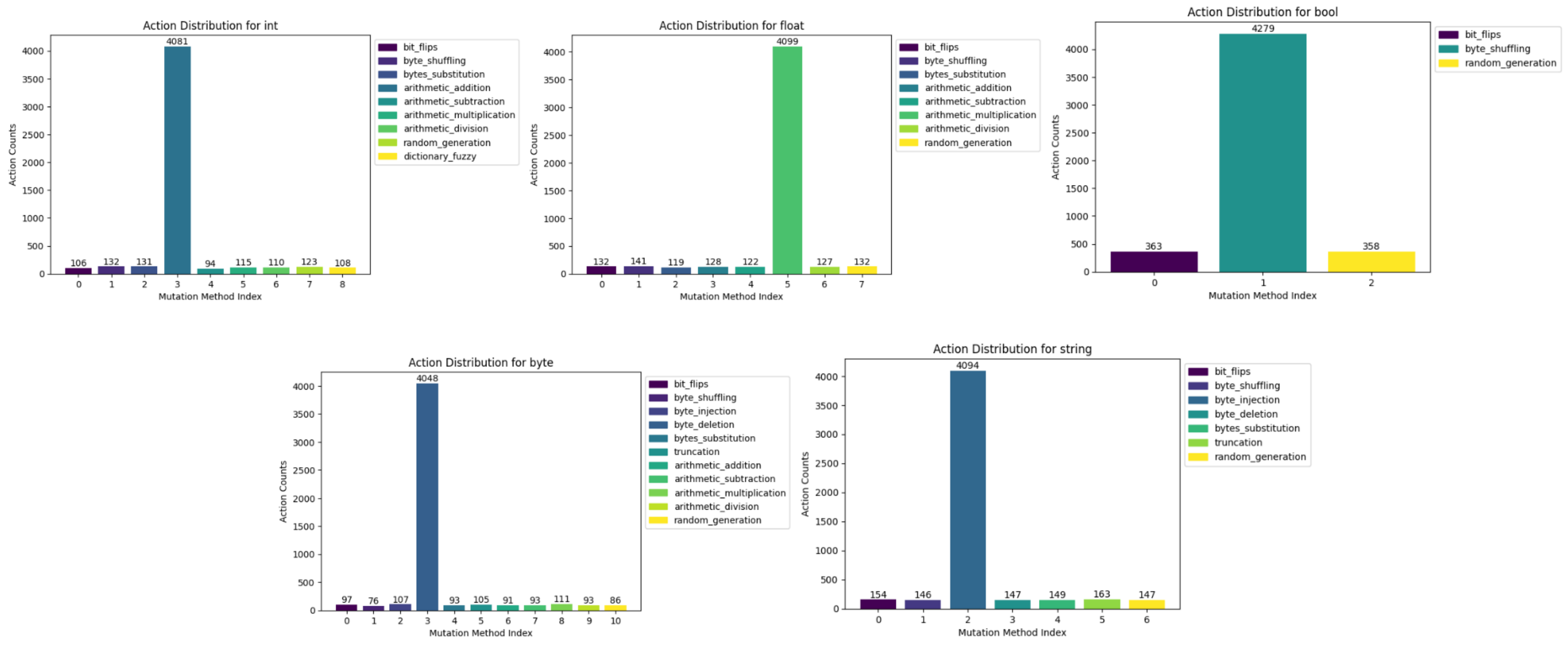


Figure 38 - UploadFile Agent’s Action Distribution

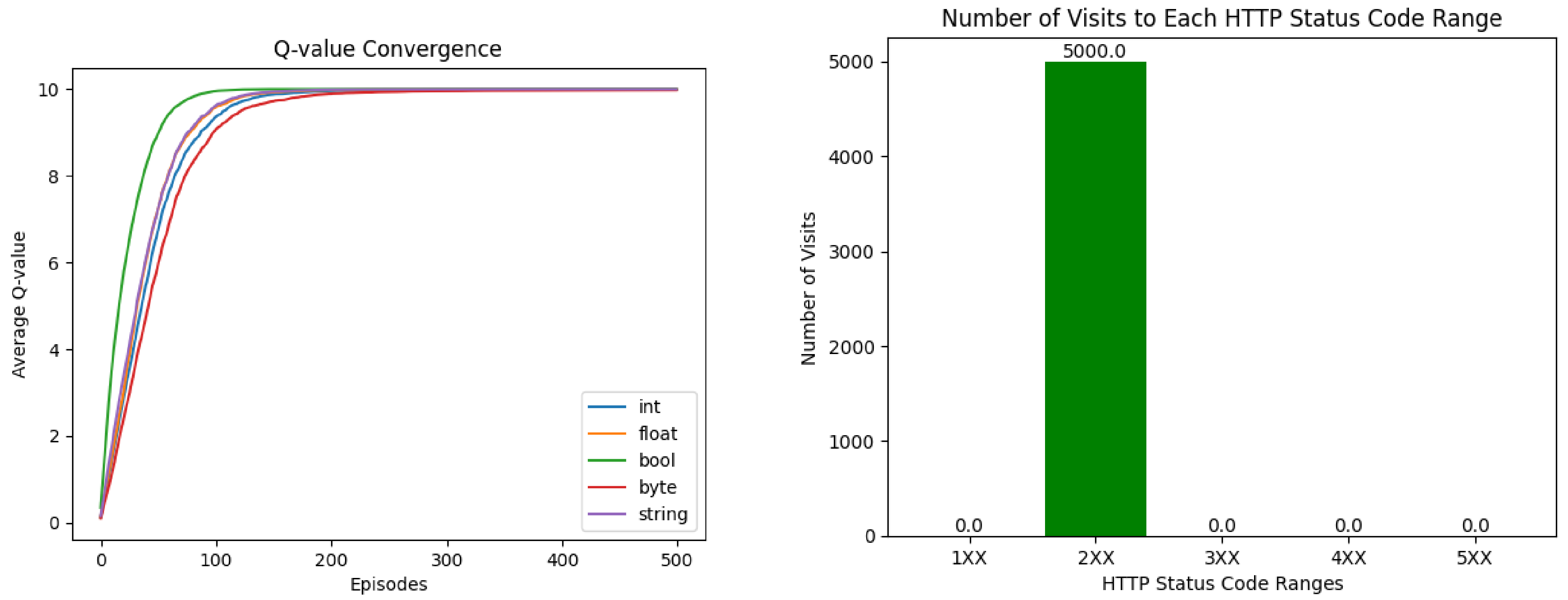


Figure 39 - UploadFile Agent’s Q-Value Convergence and State Visits

## C.7 – UpdatePetWithForm

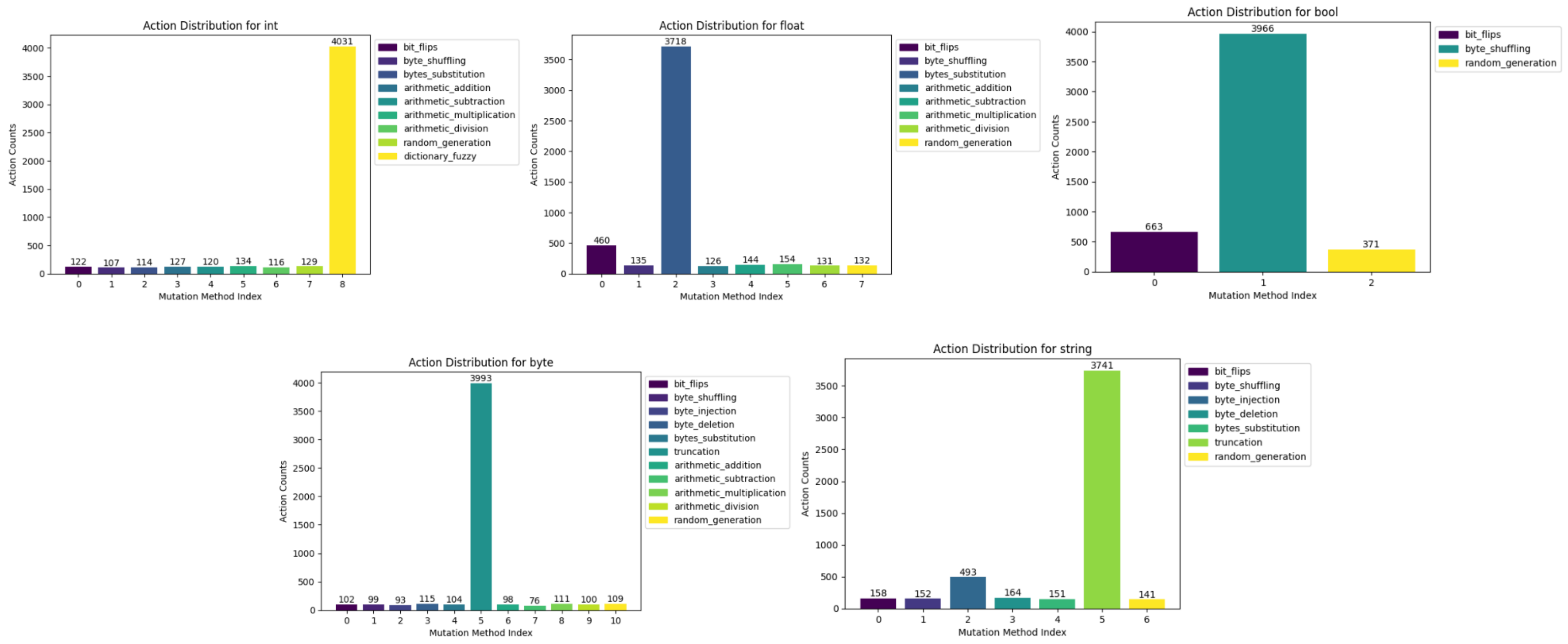
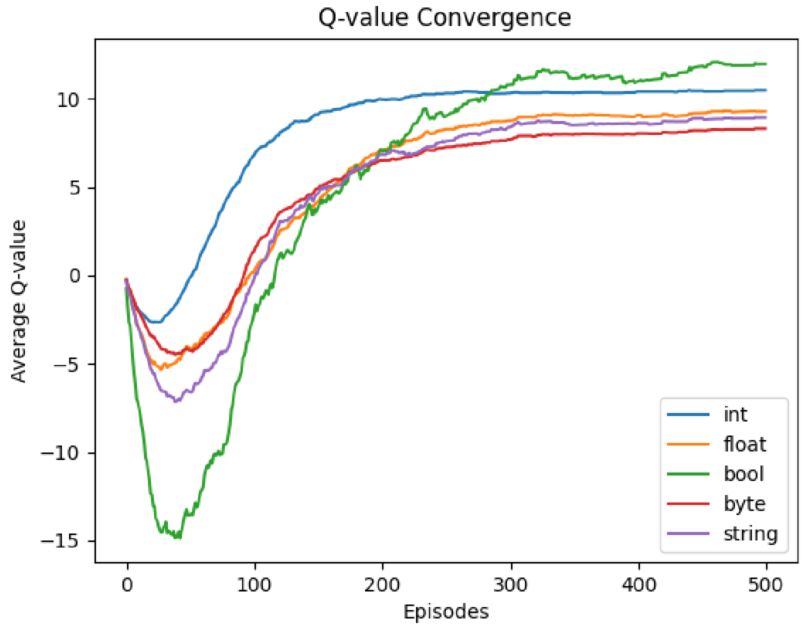


Figure 40 - UpdatePetWithForm Agent’s Action Distribution



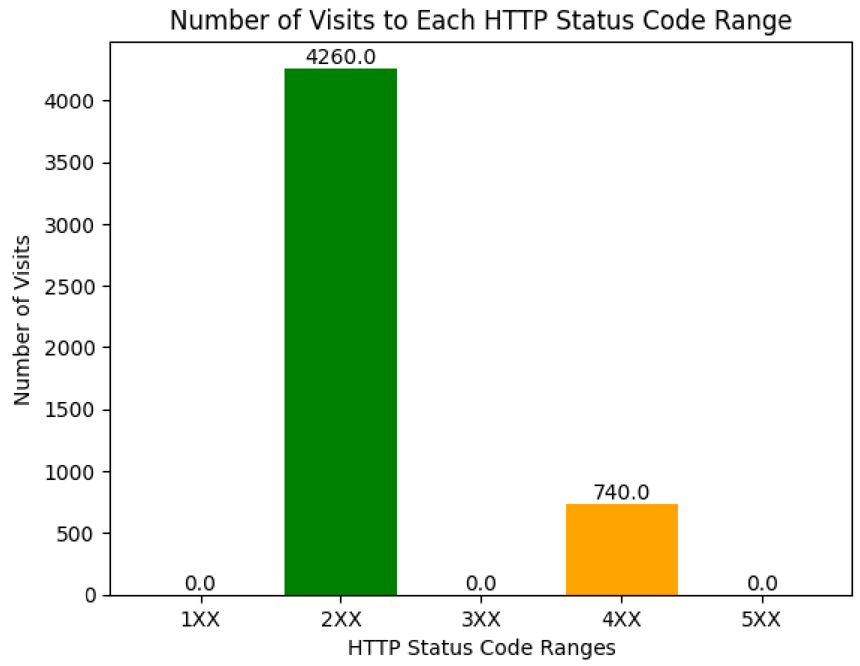


Figure 41 - UpdatePetWithForm Agent’s Q-Value Convergence and State Visits

## C.8 – DeletePet

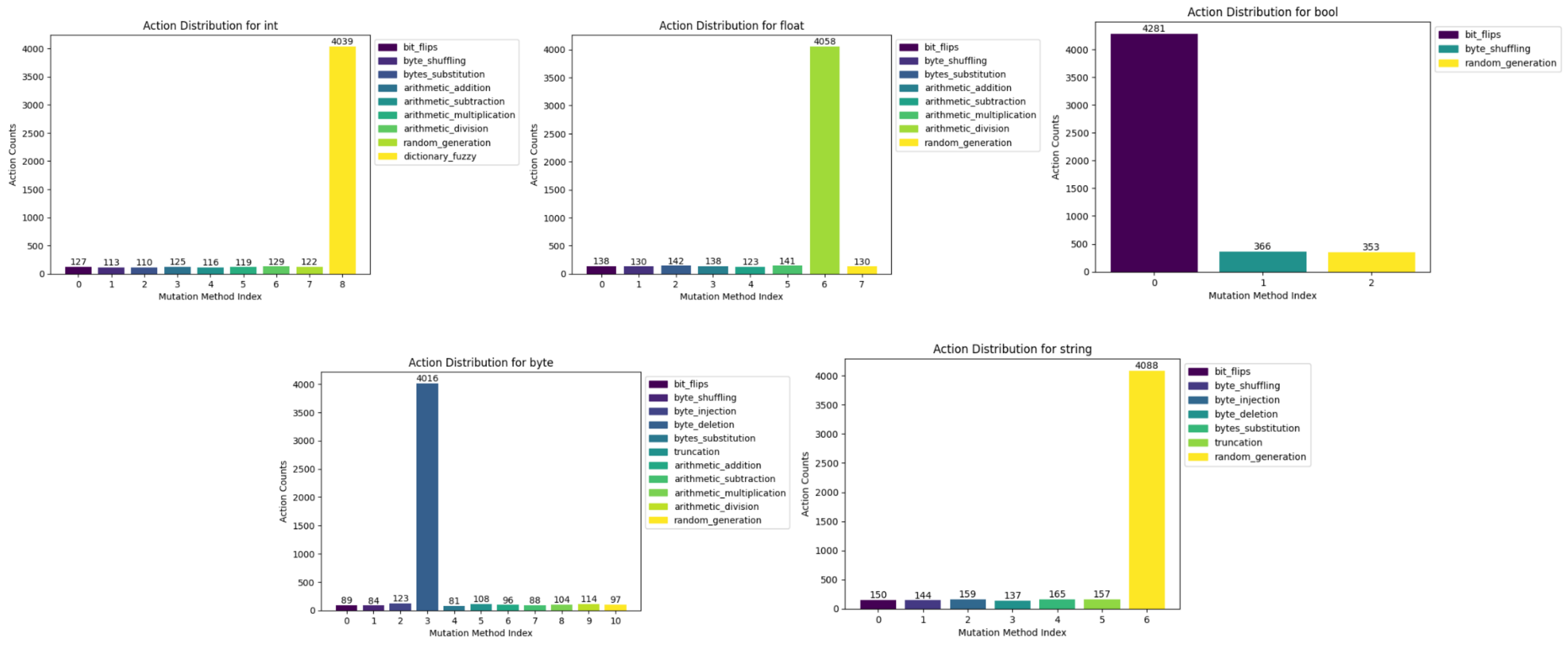
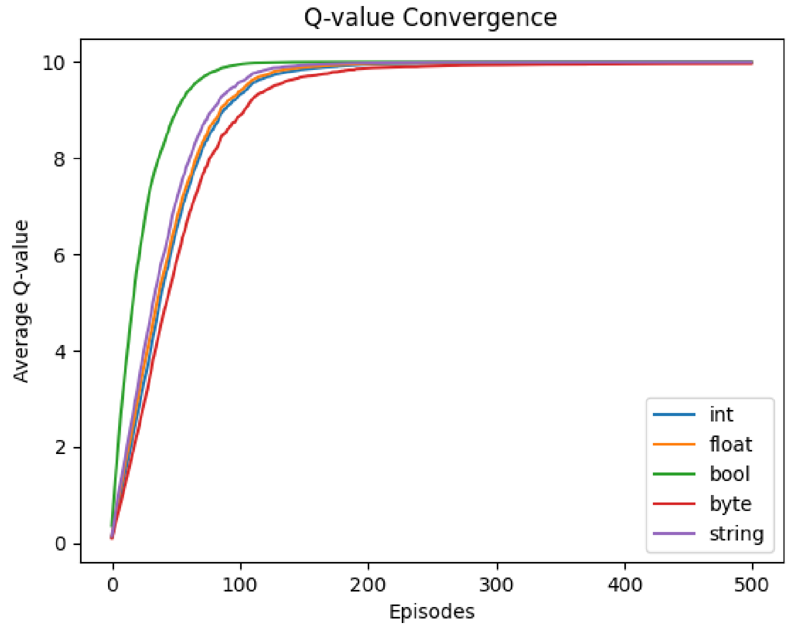


Figure 42 - DeletePet Agent’s Action Distribution



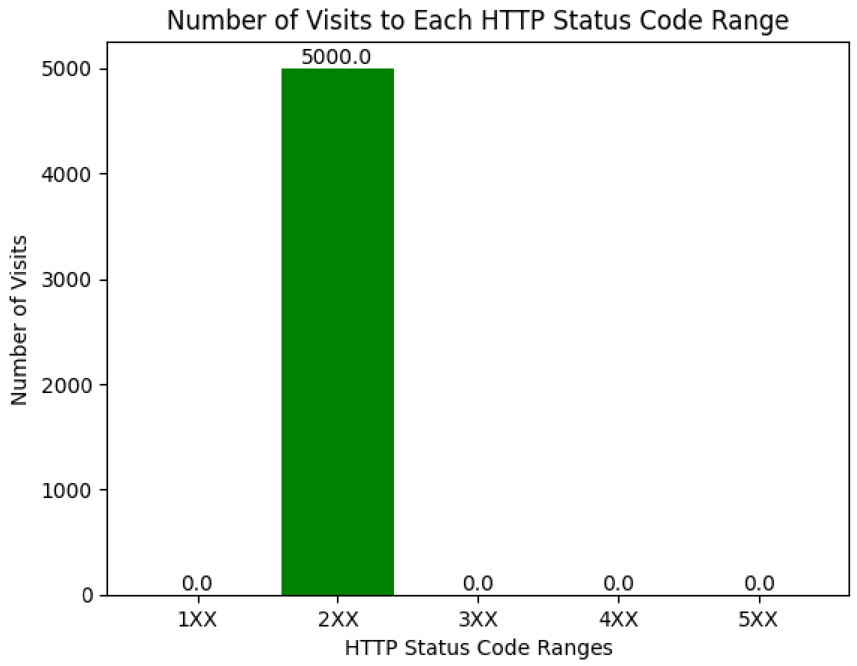


Figure 43 - DeletePet Agent’s Q-Value Convergence and State Visits

## C.9 – CreateUser

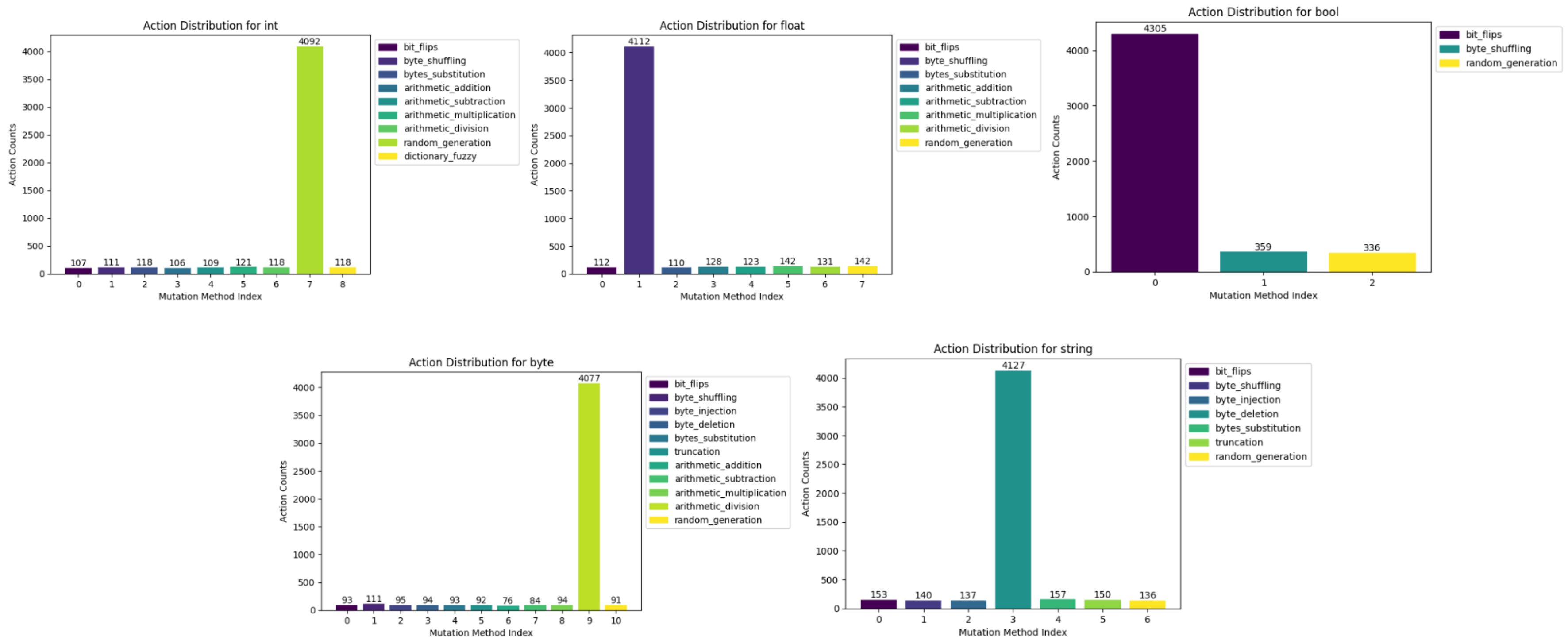
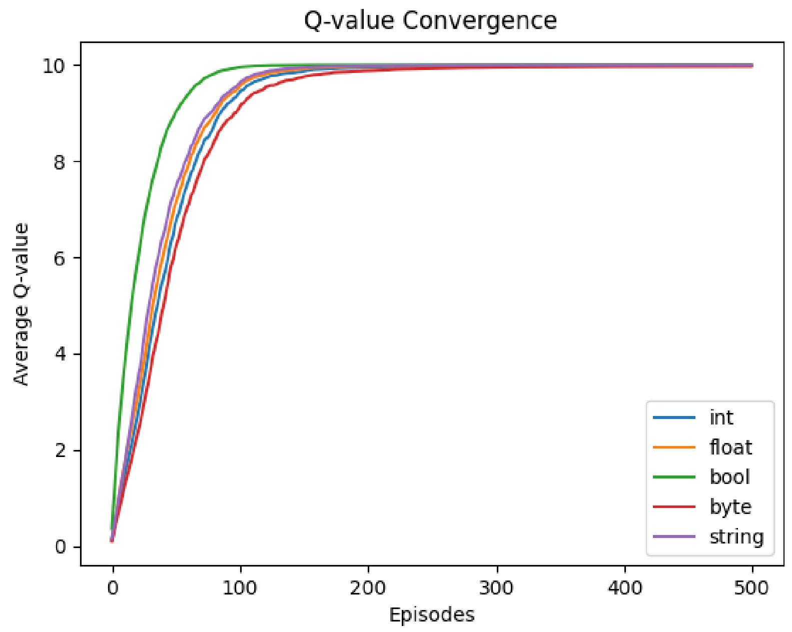


Figure 44 - CreateUser Agent’s Action Distribution



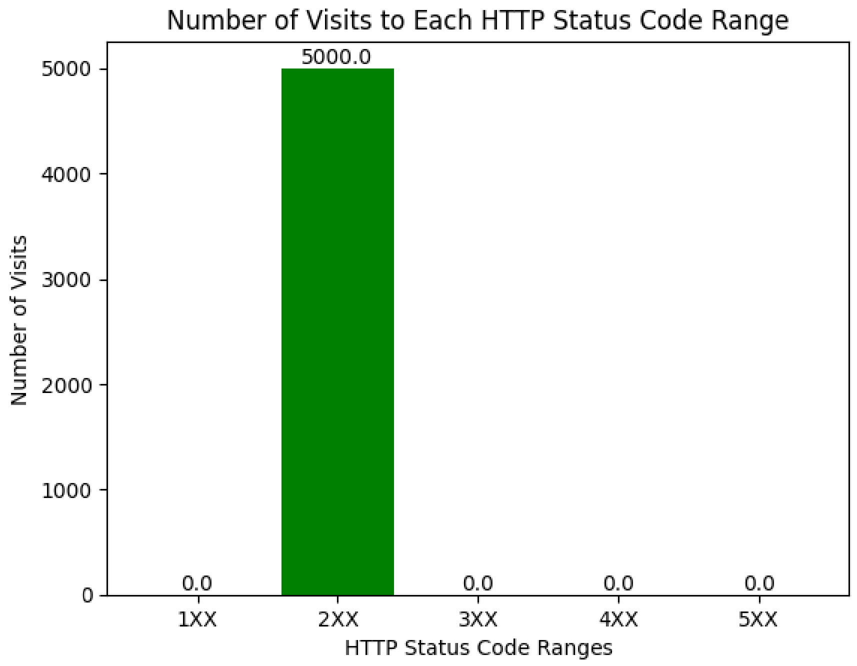


Figure 45 - CreateUser Agent’s Q-Value Convergence and State Visits

## C.10 – LoginUser

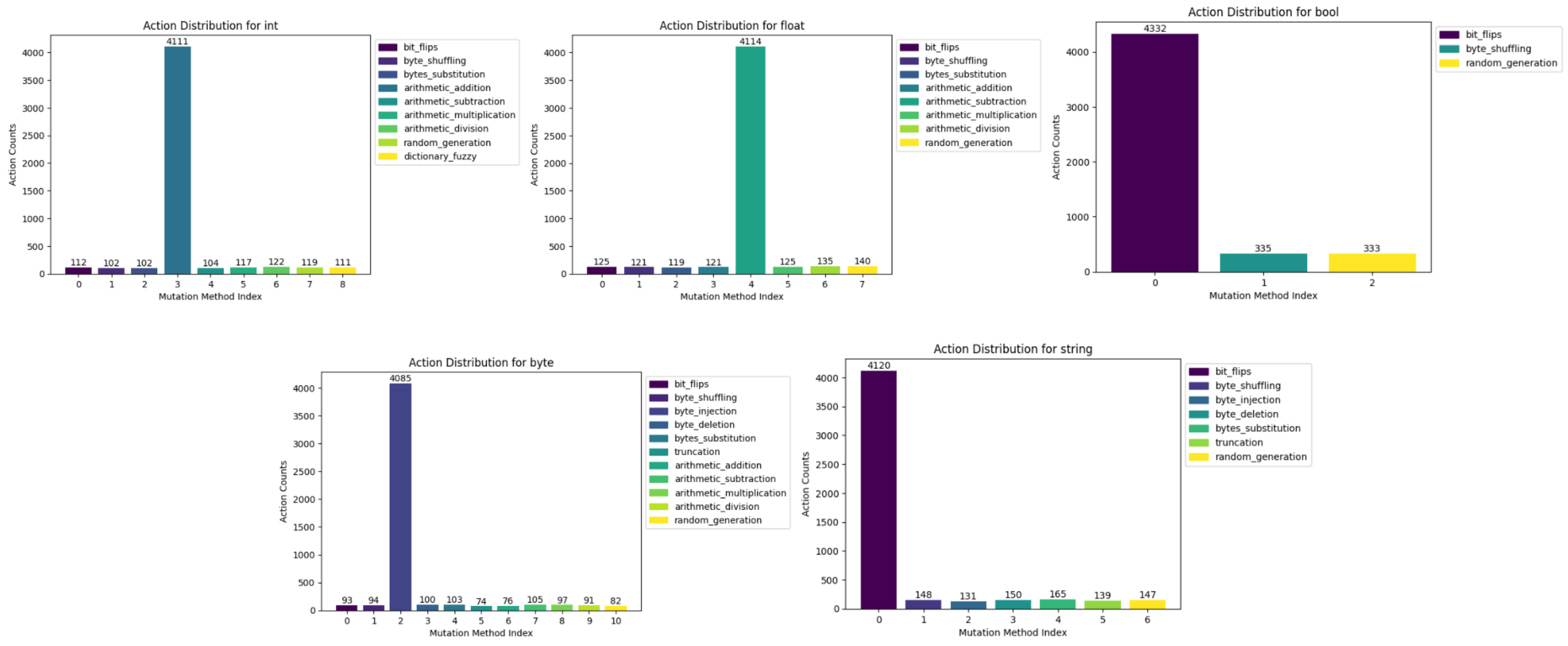
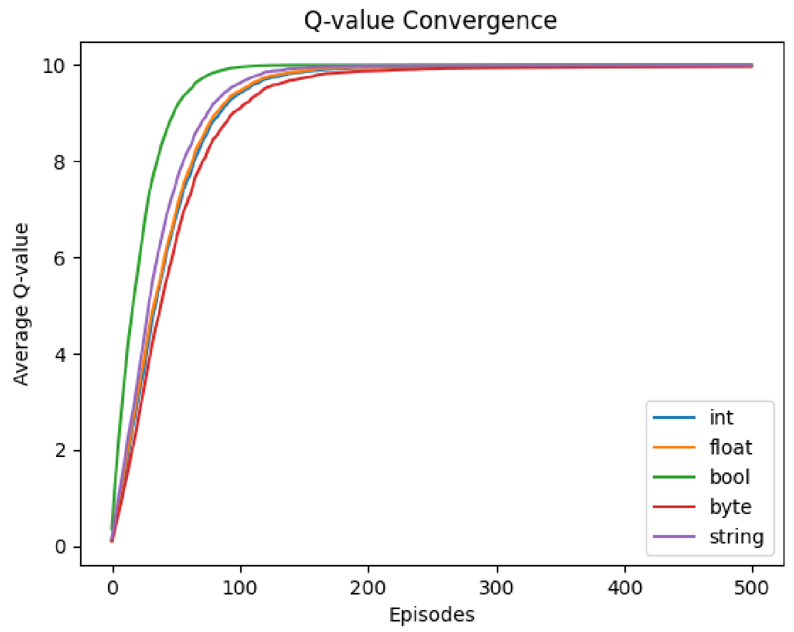


Figure 46 - LoginUser Agent’s Action Distribution



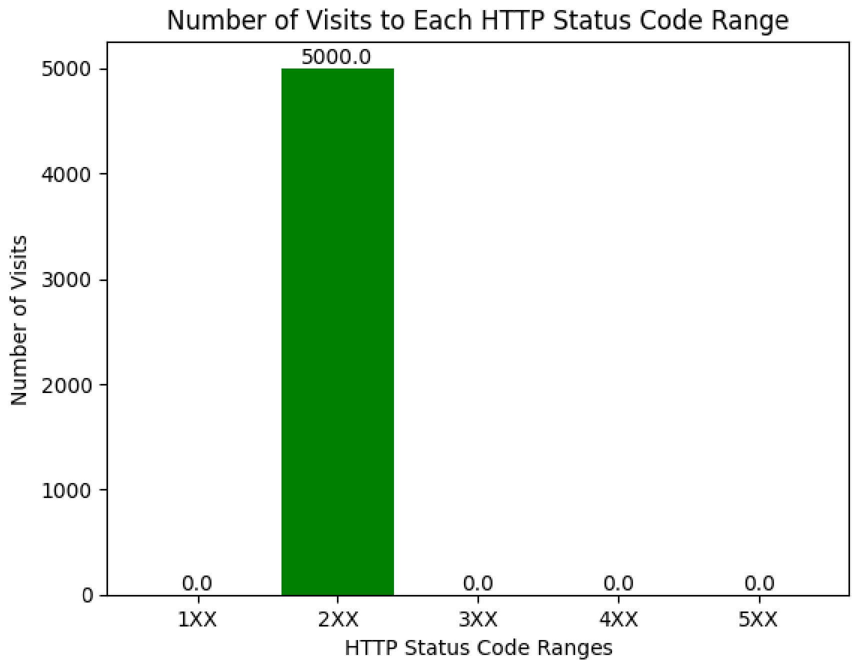


Figure 47 - LoginUser Agent's Q-Value Convergence and State Visits

## C.11 – UpdateUser

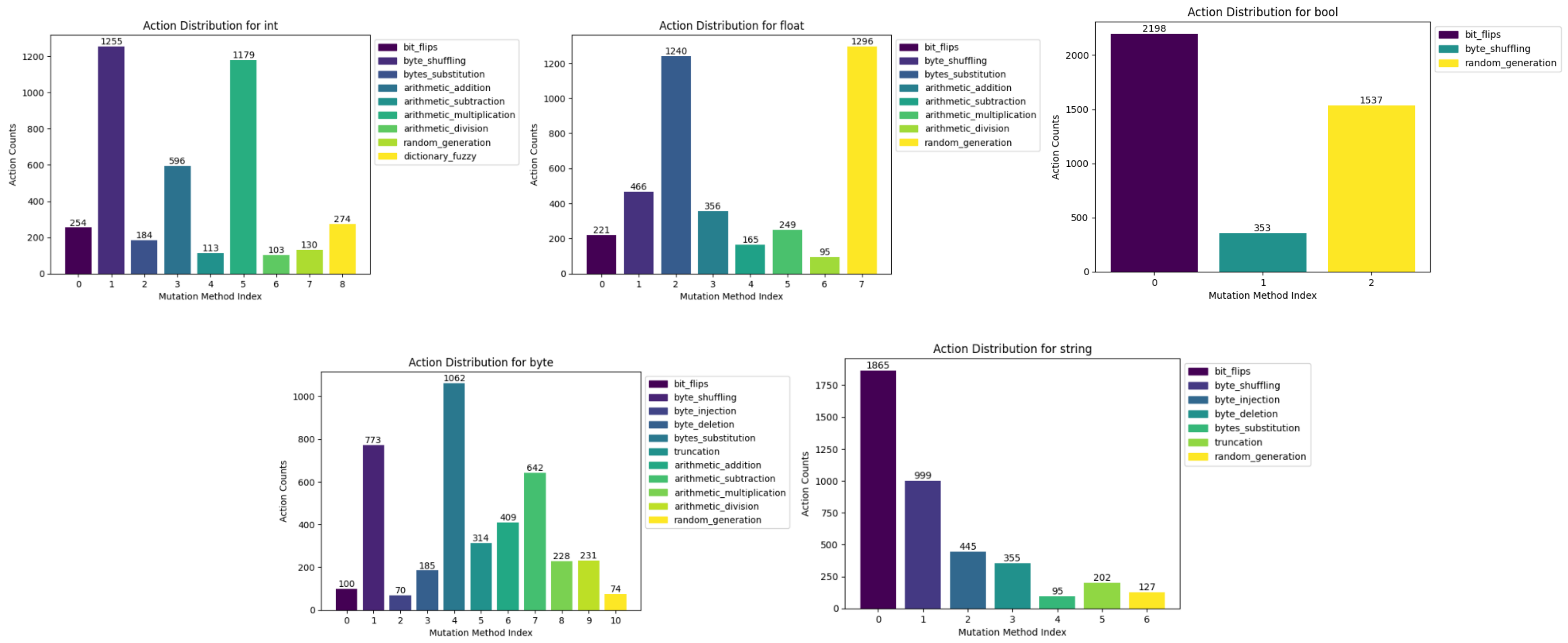
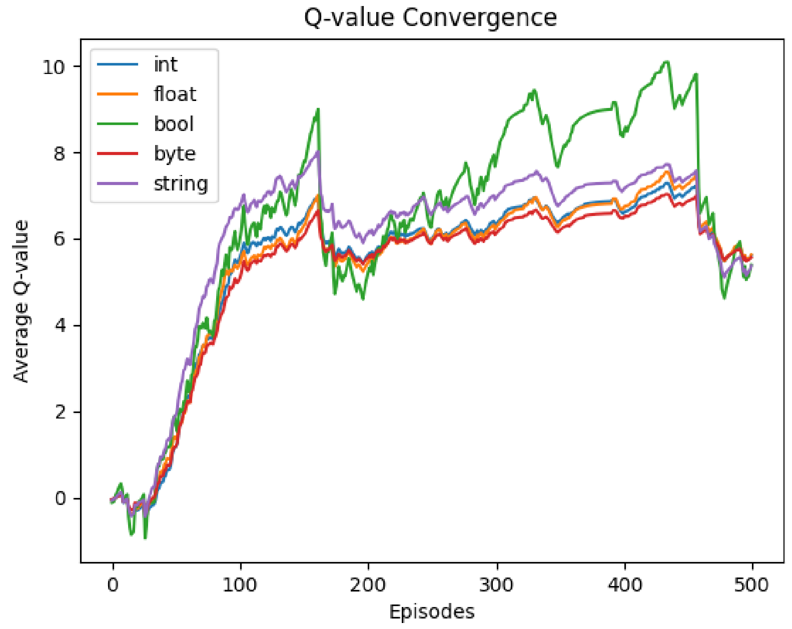


Figure 48 - UpdateUser Agent’s Action Distribution



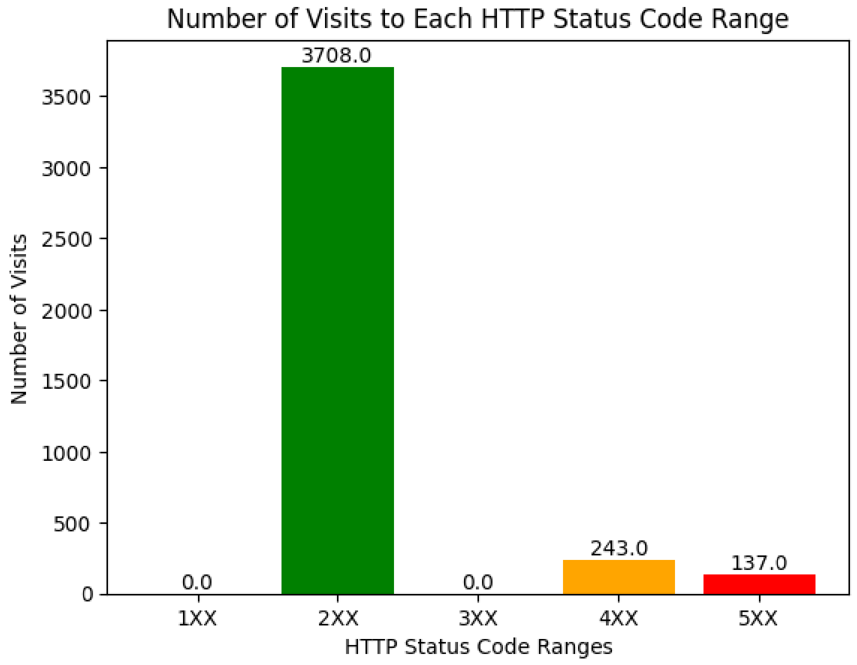


Figure 49 - UpdateUser Agent’s Q-Value Convergence and State Visits

## C.12 – LogoutUser

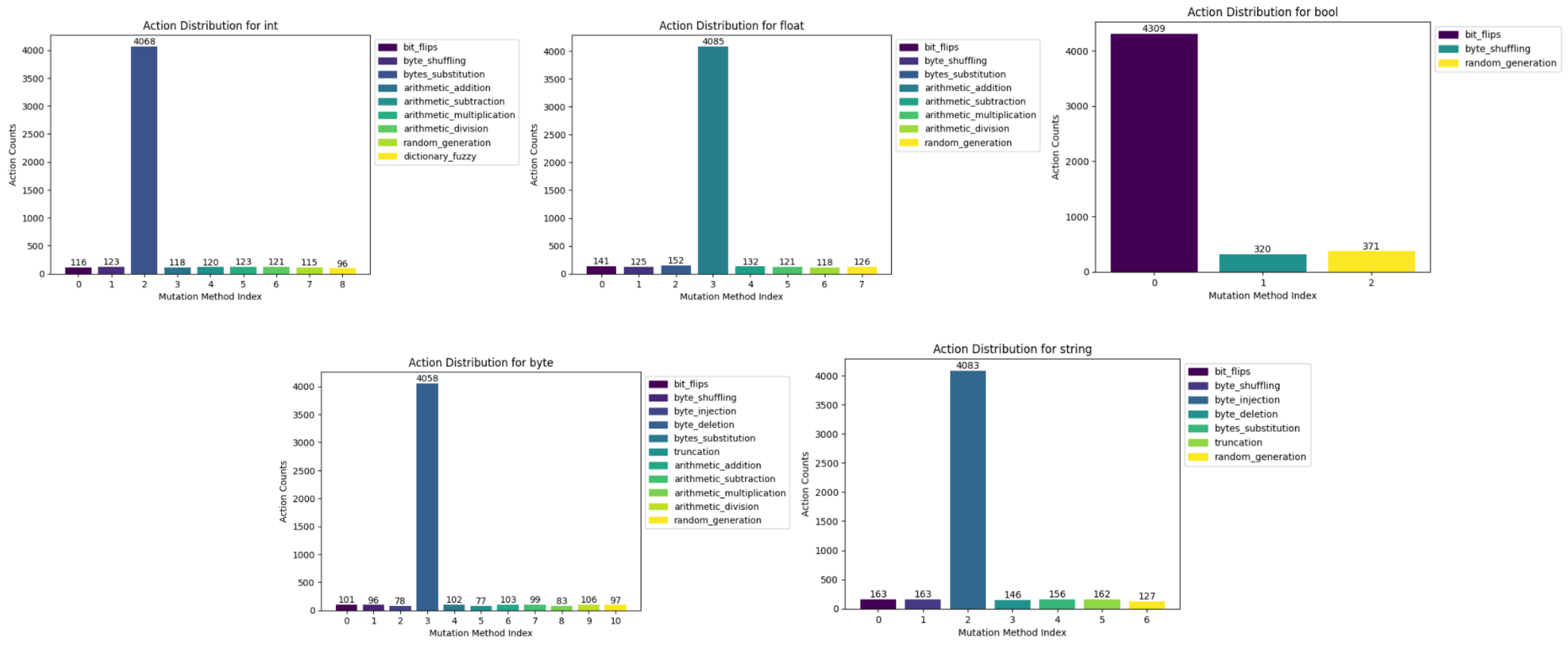


Figure 50 - LogoutUser Agent’s Action Distribution

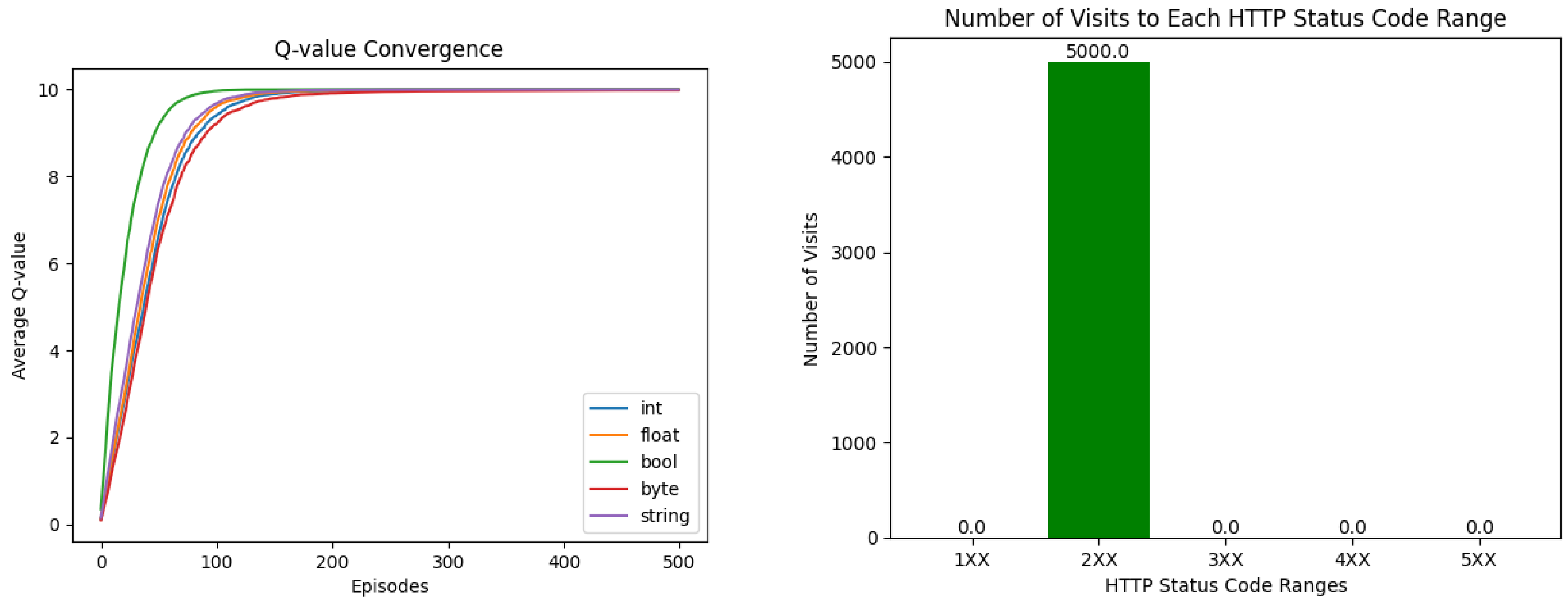


Figure 51 – LogoutUser Agent’s Q-Value Convergence and State Visits

## C.13 – GetUserByName

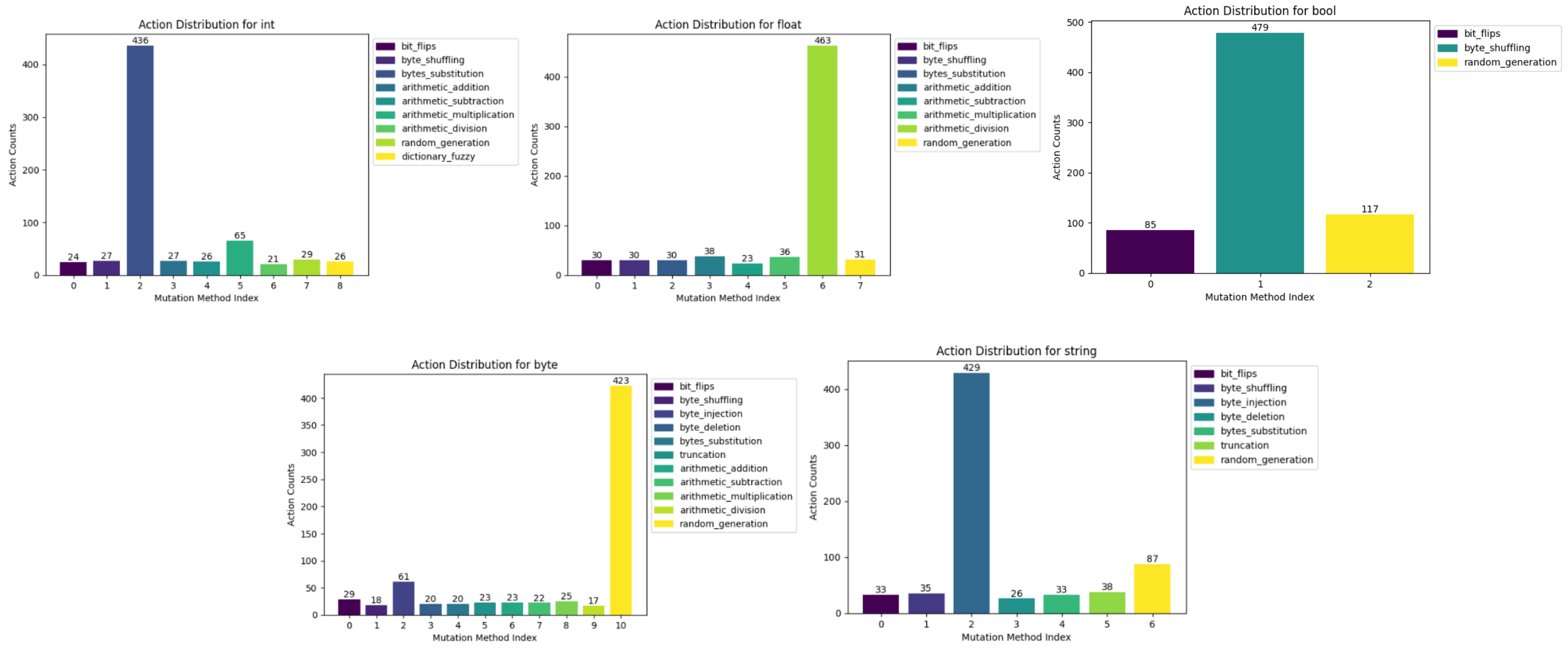


Figure 52 - GetUserByName Agent's Action Distribution

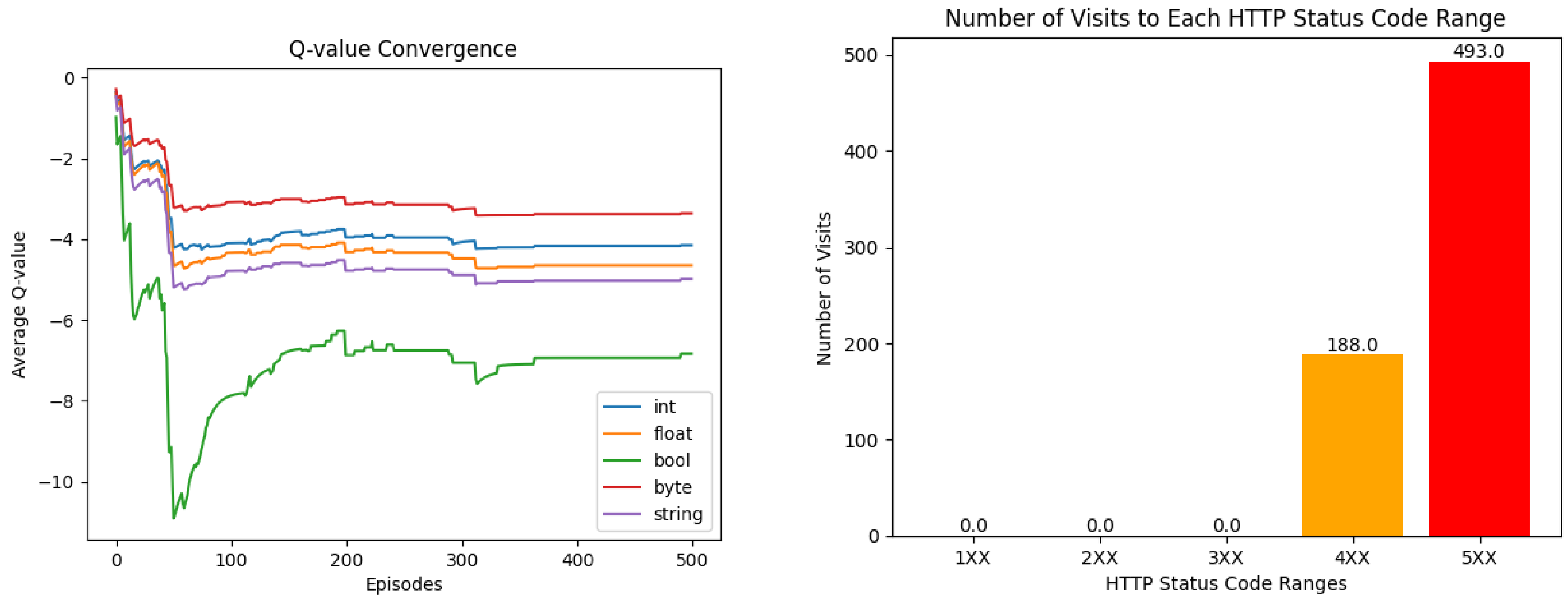


Figure 53 – GetUserByName Agent's Q-Value Convergence and State Visits

## C.14 – DeleteUser

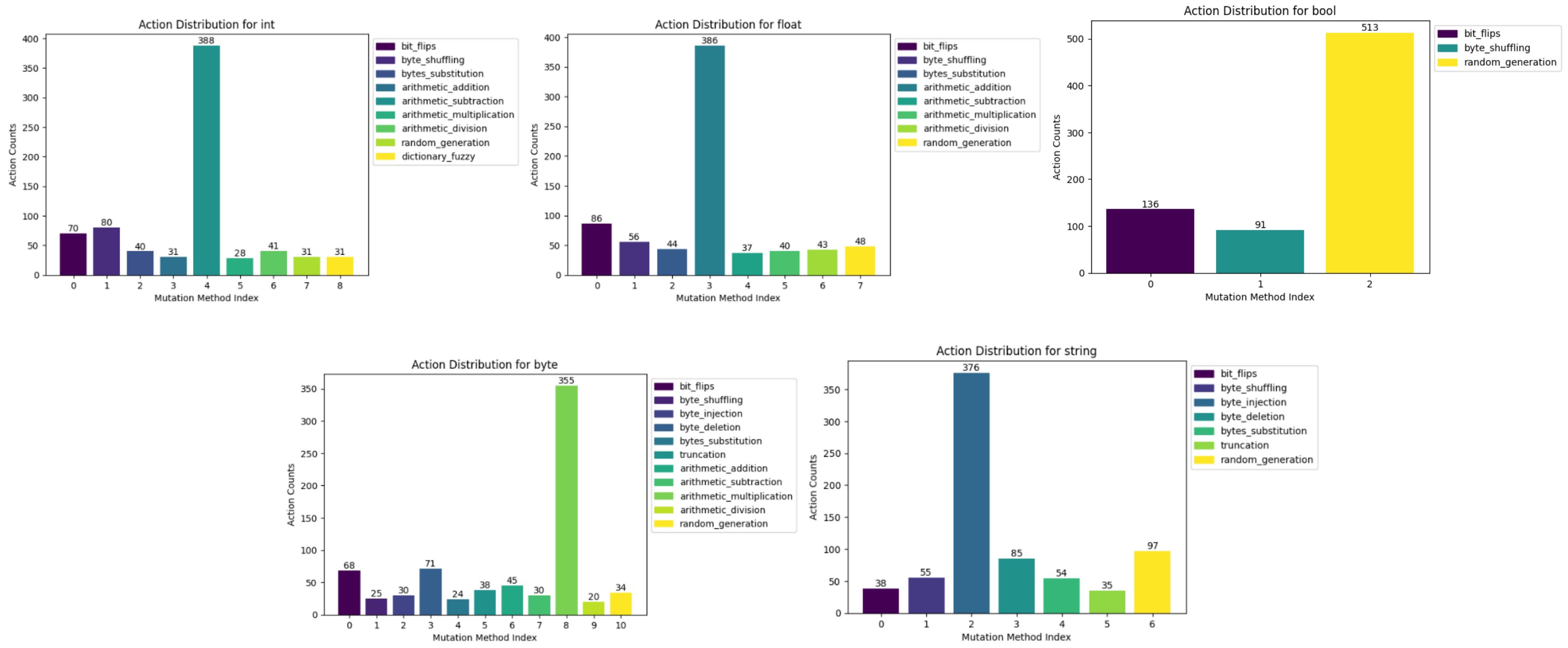


Figure 54 - DeleteUser Agent’s Action Distribution

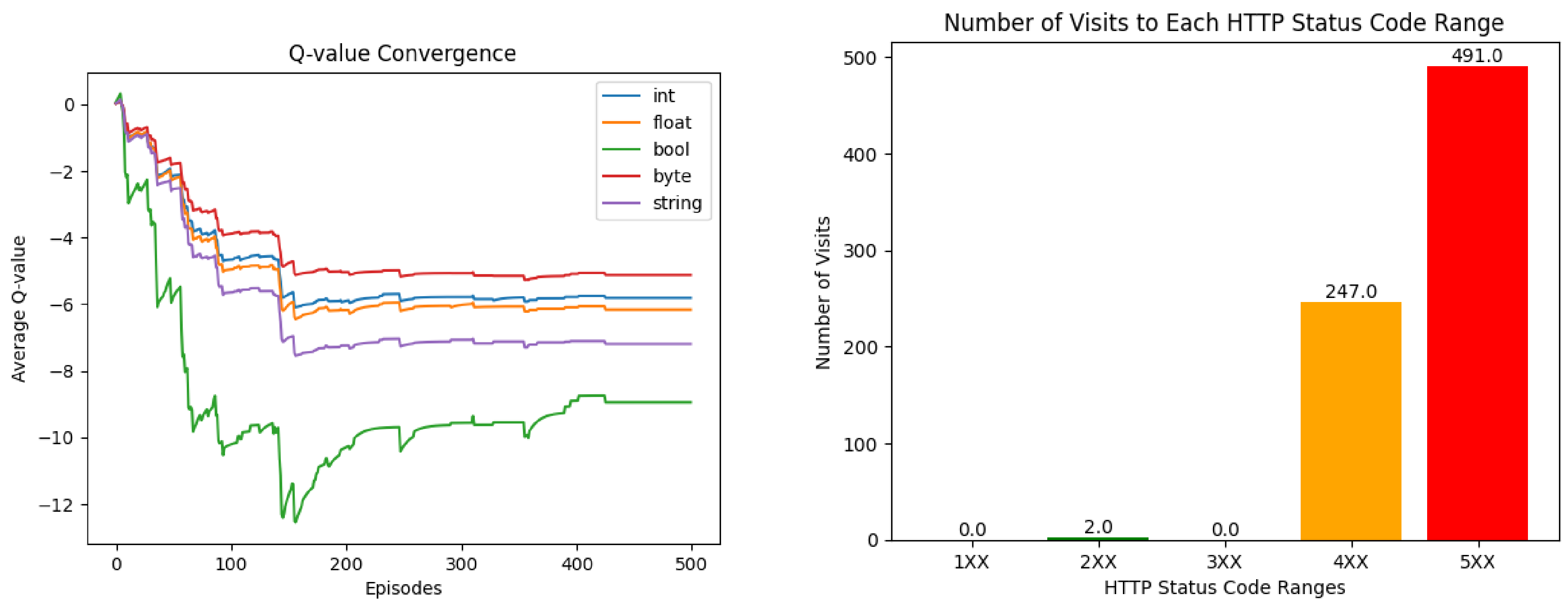


Figure 55 - DeleteUser Agent's Q-Value Convergence and State Visits

## C.15 – PlaceOrder

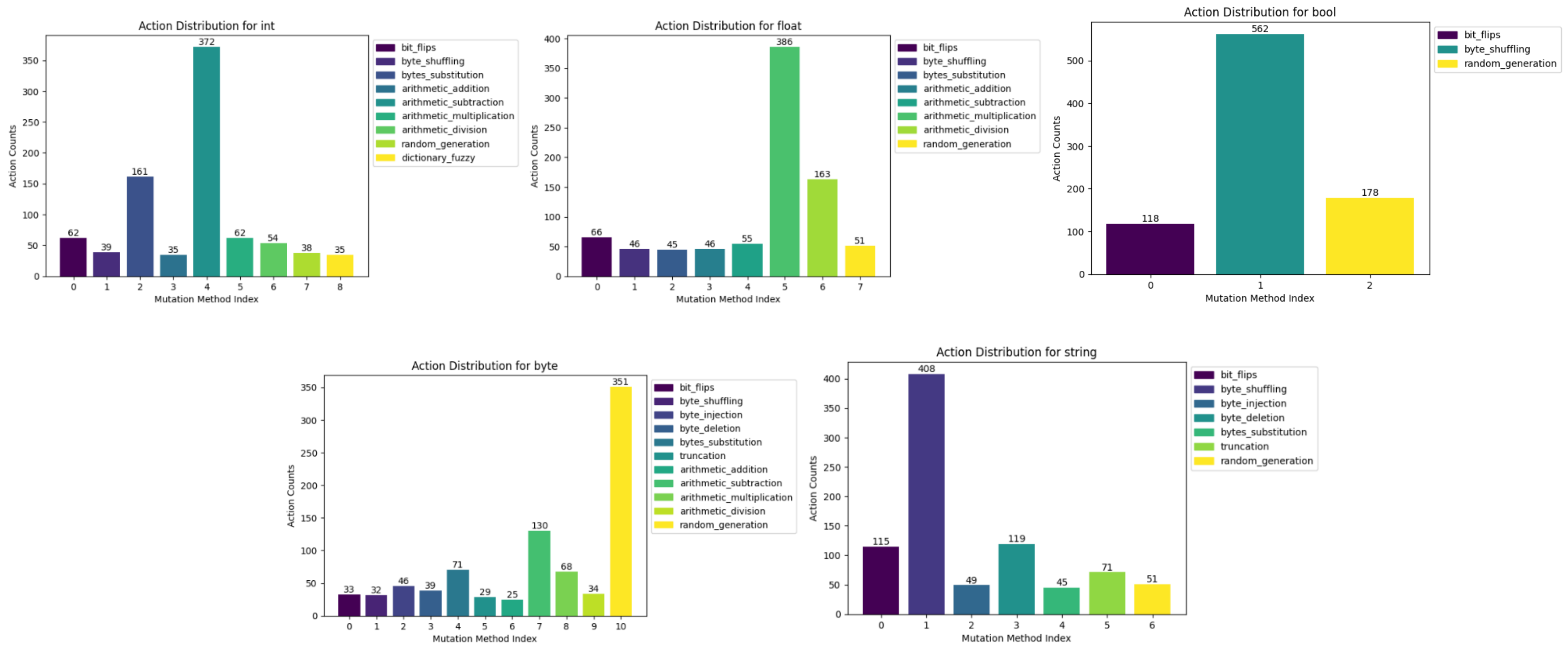


Figure 56 - PlaceOrder Agent’s Action Distribution

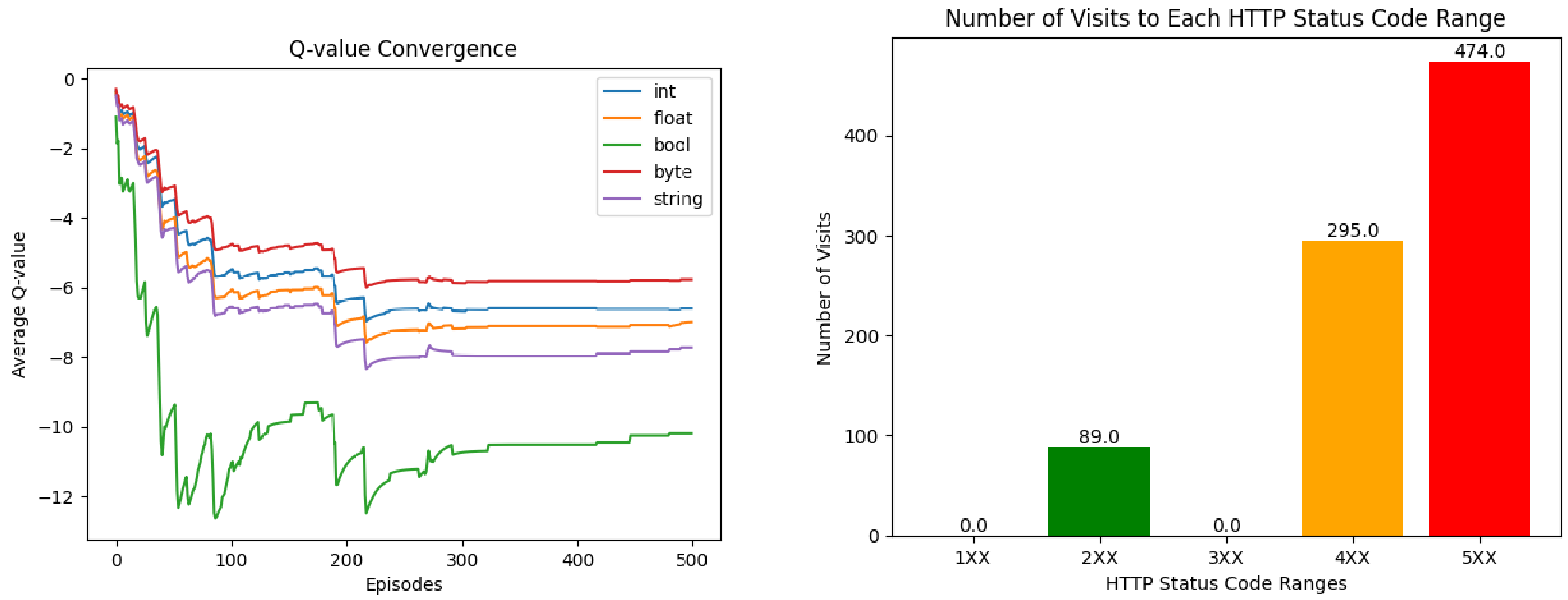


Figure 57 – PlaceOrder Agent's Q-Value Convergence and State Visits

## C.16 – GetOrderById

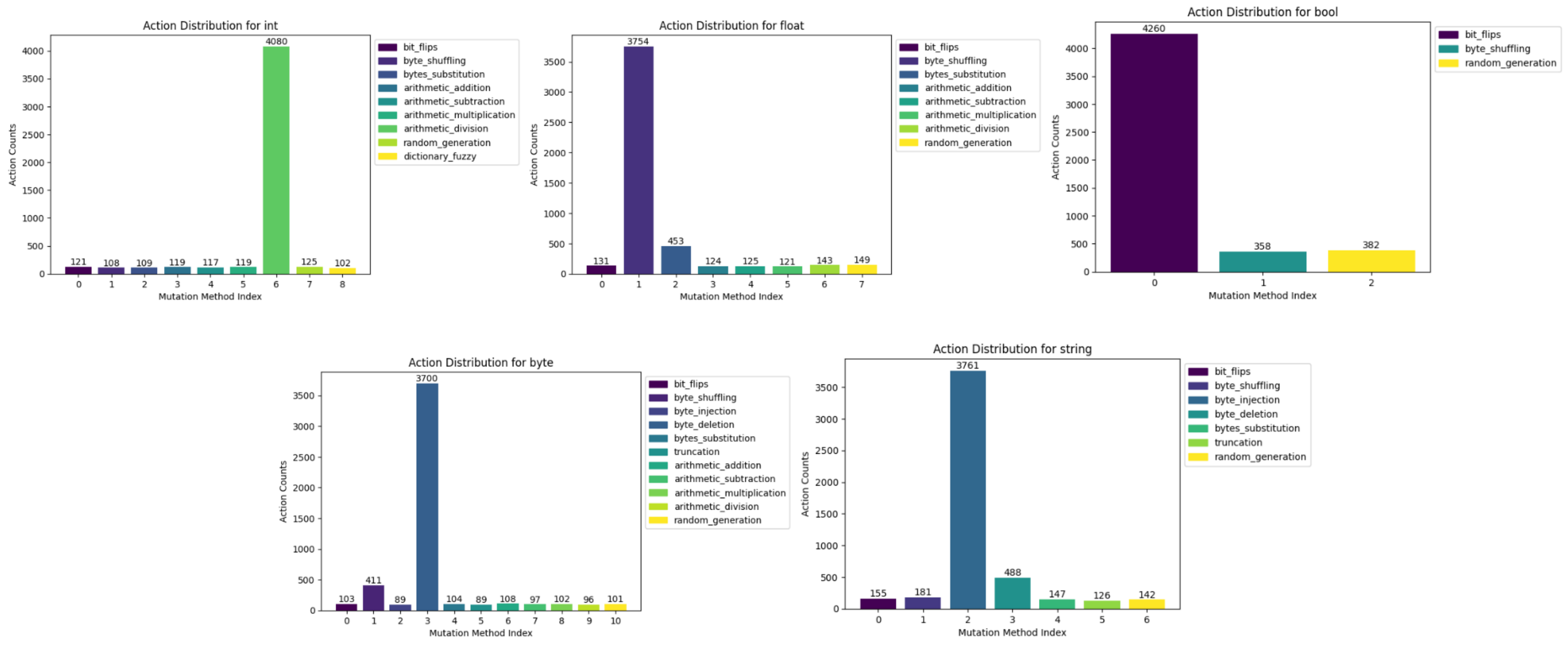


Figure 58 - GetOrderById Agent's Action Distribution

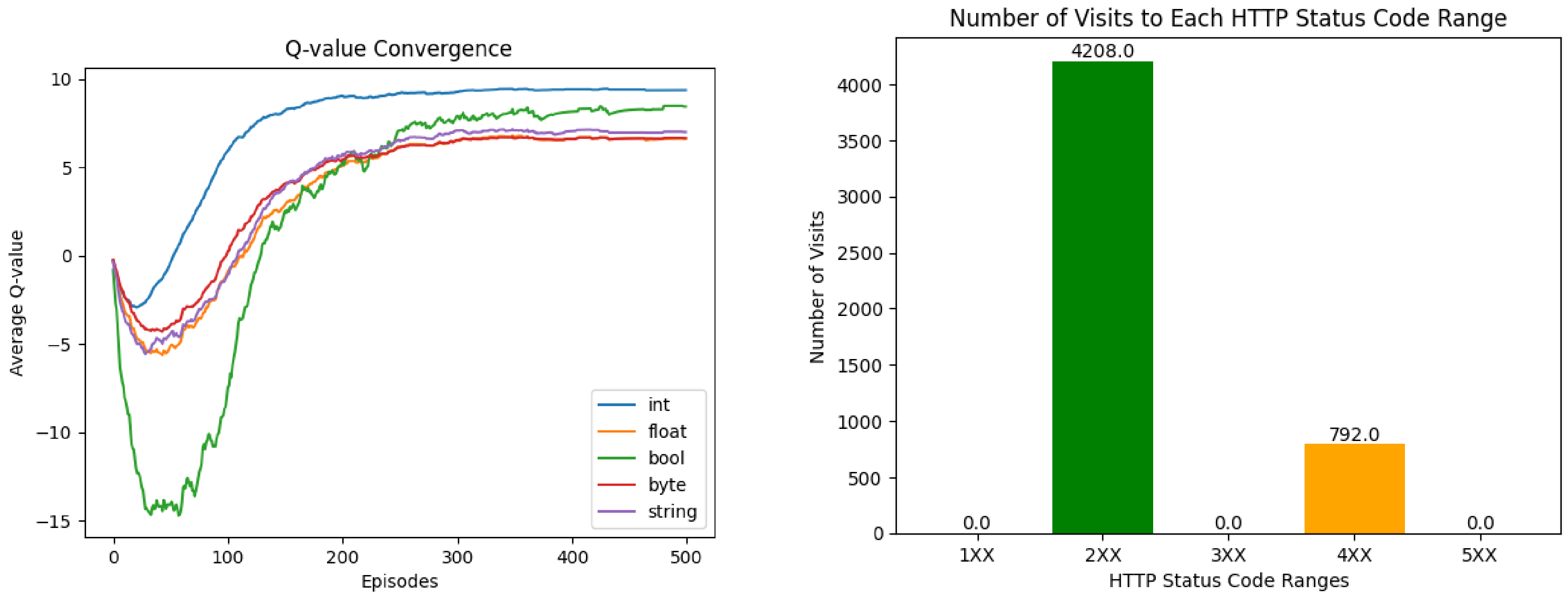


Figure 59 - GetOrderById Agent's Q-Value Convergence and State Visits

## C.17 – GetInventory

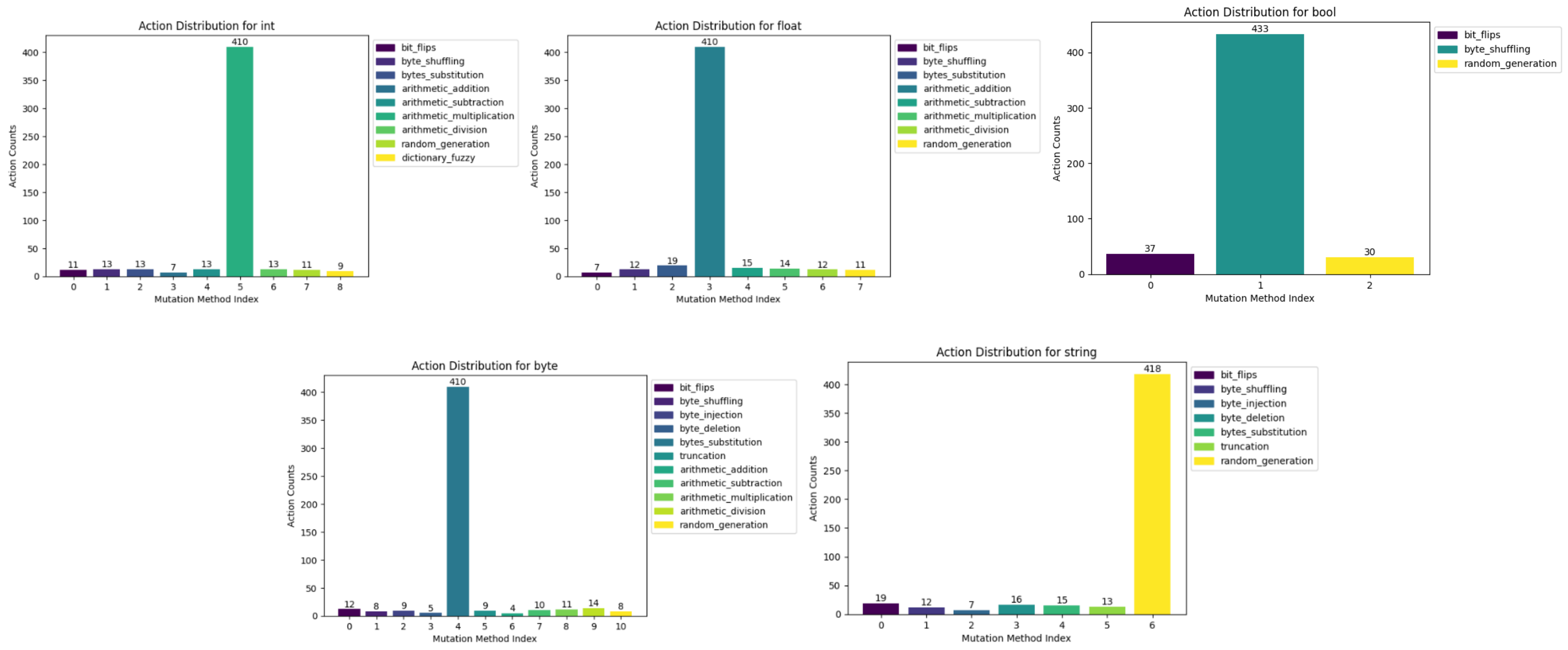
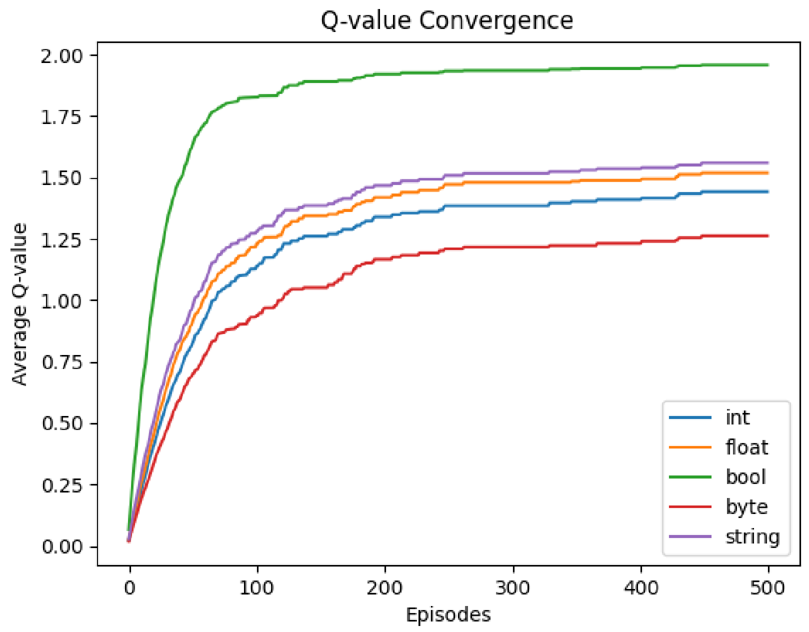


Figure 60 – GetInventory Agent’s Action Distribution



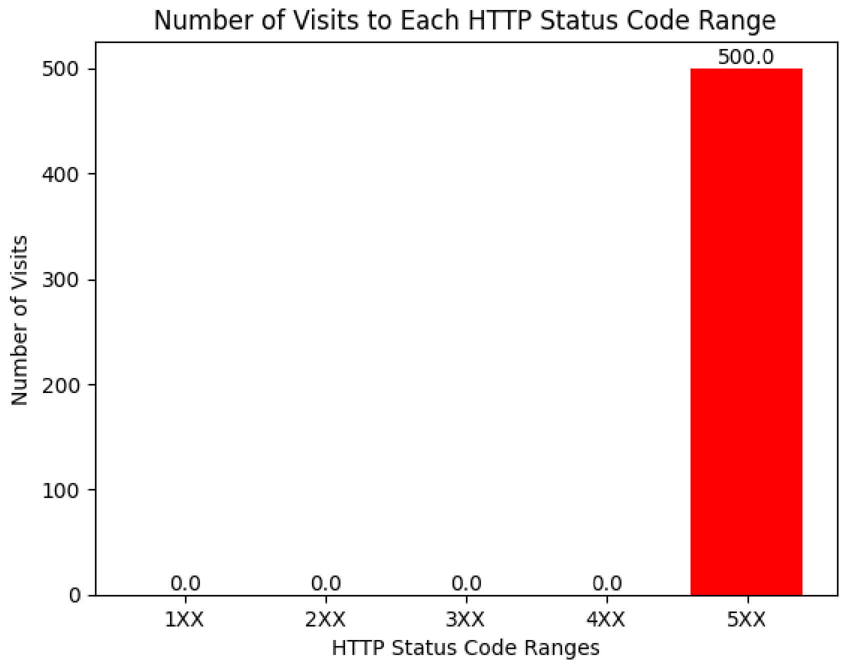


Figure 61 - GetInventory Agent’s Q-Value Convergence and State Visits

## C.18 – DeleteOrder

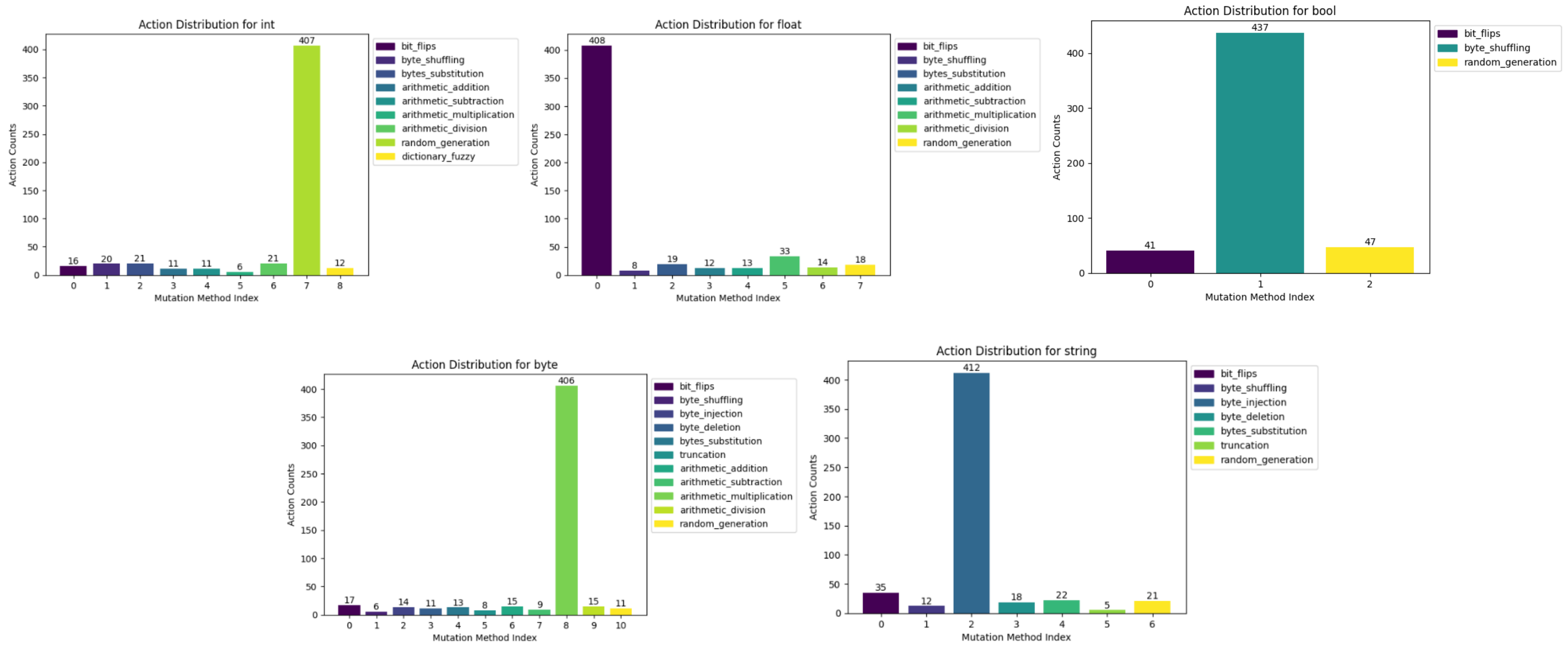
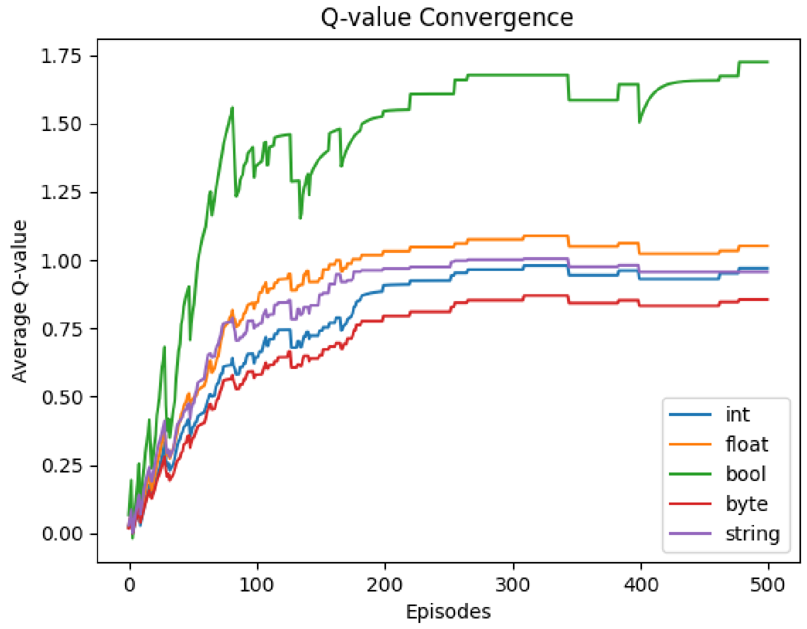


Figure 62 - DeleteOrder Agent’s Action Distribution



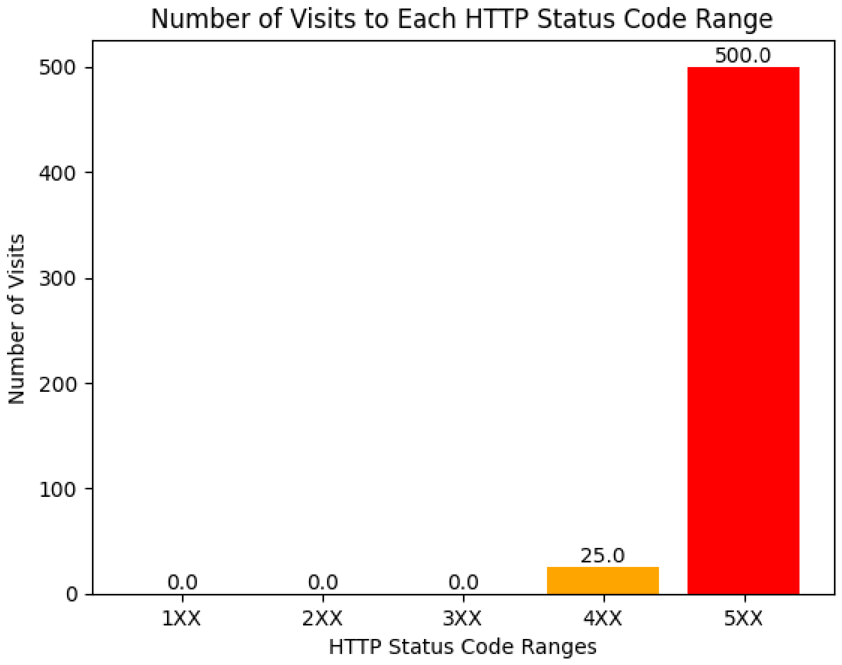


Figure 63 - DeleteOrder Agent's Q-Value Convergence and State Visit

## Appendix D – Status Codes Function Distribution

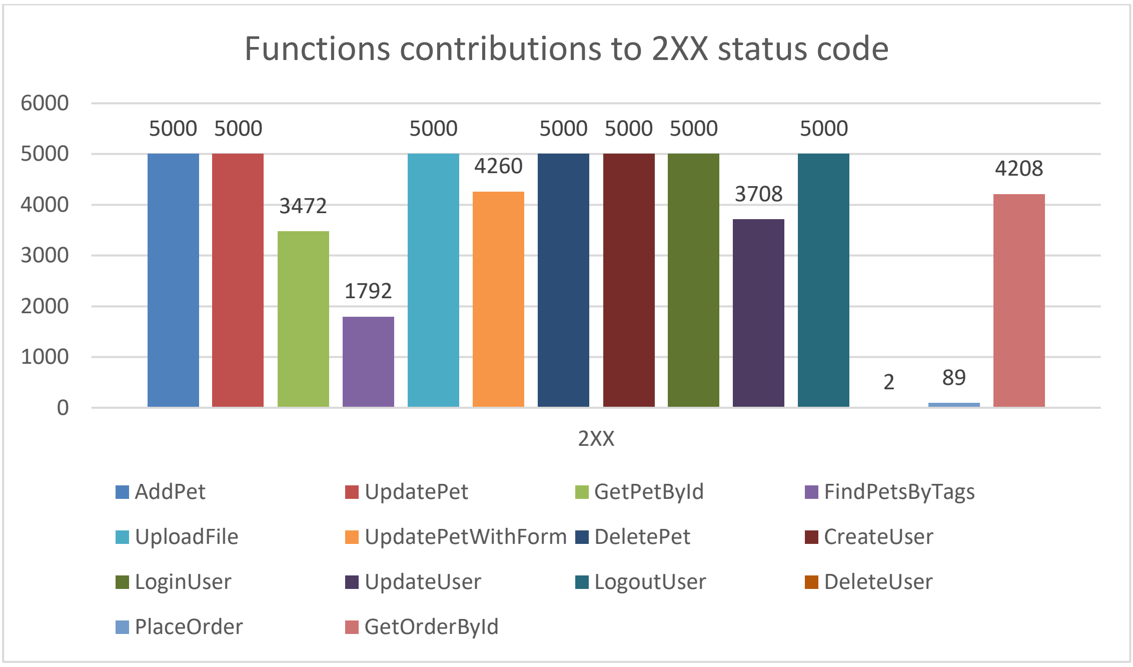


Figure 64 - Functions Contributions to 2XX Status Code

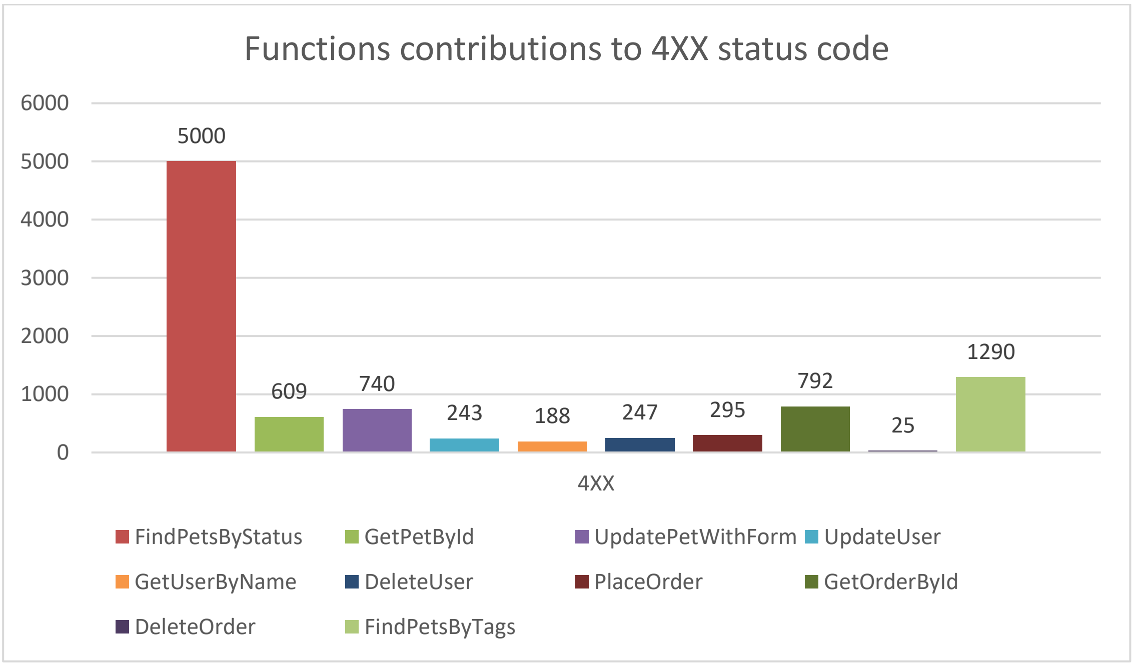


Figure 65 - Functions Contributions to 4XX Status Code

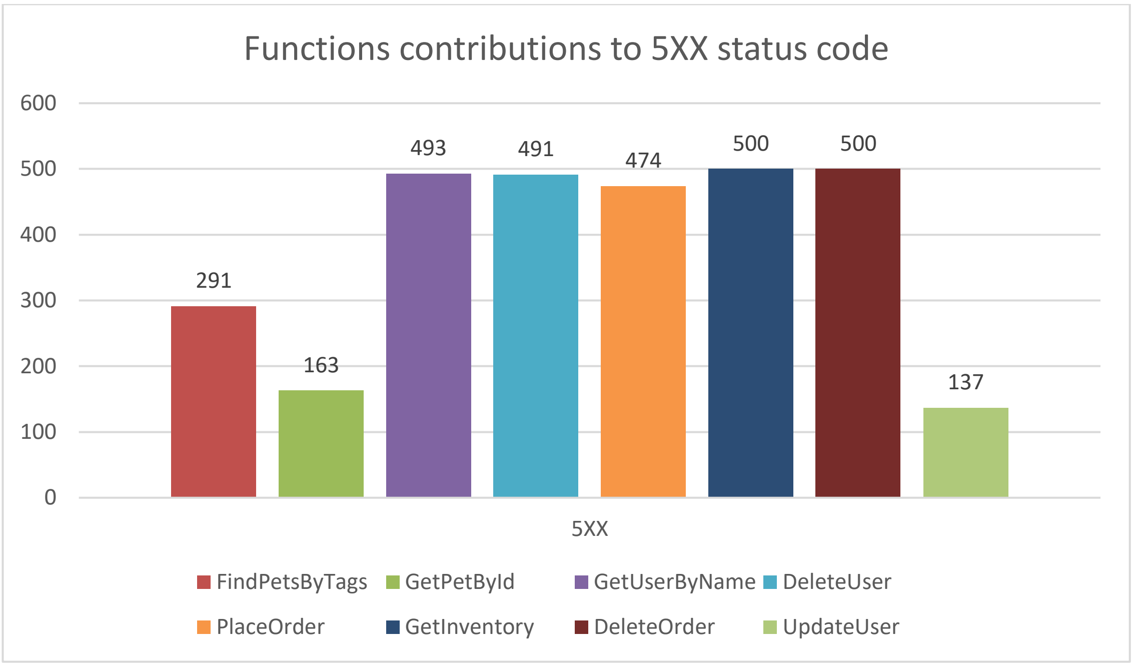


Figure 66 - Functions Contributions to 5XX Status Code