National College of Ireland

AI-Driven Test Case Generation and Optimization

MSc Research Project
Cloud Computing

Mohammad Saif
Student ID: 22248218

School of Computing
National College of Ireland

Supervisor: Aqeel Kazmi

National College of Ireland

MSc Project Submission Sheet

School of Computing

Student Name: ……. Mohammad Saif ……

Student ID: …….22248218……

Programme: …MSc in Cloud Computing…… **Year:** ……2024……..

Module: ……MSc Research Project……..……

Supervisor: ……. Aqeel Kazmi……..……

Submission Due Date: ……16th September 2024……..……

Project Title: …AI-Driven Testcase generation and Optimization……..……

Word Count: ……7099…… **Page Count** ……21…..……..…..

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.
ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature:  ……

Date: ……16th September 2024……

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST

Attach a completed copy of this sheet to each project (including multiple copies)	□
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	□
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	□

Assignments that are submitted to the Programme Coordinator Office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

AI-Driven Testcase Generation and Optimization

Mohammad Saif
22248218

Abstract

The aim of this research is to develop an AI-Driven model to enhance efficiency and effectiveness of software testing by generating and ordering testcases using Natural language Processing (NLP) and Reinforcement Learning (RL) techniques. The traditional software testing methods are time-consuming and require significant manual effort, which often leads to inefficiency and missing test coverage. This study utilizes NLP to automatically extract test cases from software requirements documents and applies RL to order test execution sequence. The integration of these technologies aims to maximize test coverage, improve testing efficiency and saving time. Through automated test case generation and optimization, this research aims to reduce test execution time and enhance test coverage, thereby supporting more reliable and efficient software development practices. The findings from this study highlight the potential impact of combining NLP and RL in automating software testing process, promising substantial improvements in software quality assurance and development workflows.

Keywords— AI, ML, NLP, RL, Test cases, DQN, A2C, SRS

1 Introduction

1.1 Background

The rapid digitization and increase in complexity of software systems have led to the emergence of more complex testing methodologies. Tradition manual test case generation methods are often time-consuming and error-prone, leading to inefficiencies and increased costs. With software systems becoming more complex, the demand for rigorous testing that ensures reliability and performance has never been more critical. The evolution of Artificial Intelligence (AI) presents an opportunity to revolutionize this domain by automating and optimizing test case generation. AI technologies can enhance software testing by reducing manual effort, improving accuracy and eliminating human errors therefore addressing the current inefficiencies in software testing processes.

1.2 Motivation

The importance of this research lies in utilizing the potential of AI technologies to transform software testing practices. By leveraging AI, specifically Natural Language Processing (NLP) and Reinforcement Learning (RL), this research aims to create an accurate, streamlined, and time-effective solution for test case generation. This approach is particularly important in Agile software development environments, where requirements keep on changing frequently therefore making it challenging to maintain up-to-date test cases. Traditional methods often struggle to adapt swiftly, leading to gaps and increased risk of defects in product. Using the AI-driven approach will help with substantial cost and time savings leave sufficient time to Verify the generated test cases, improving overall software quality and reliability.

1.3 Research Question

This study aims to address the following research question:

How can AI-Driven techniques AI-driven techniques such and NLP and RL, be integrated to automate and optimize test case generation and execution, thereby improving the efficiency and effectiveness of software testing process?

1.4 Research Objectives

To address the research question, the following research objectives are proposed.

- Investigate state of art AI-driven test case generation techniques
- Design a framework that utilizes NLP to parse and understand software requirements document for extracting relevant test cases.
- Develop RL algorithms to optimize sequence of test case execution, aiming to enhance test coverage and reduce execution time.
- Evaluate the effectiveness and accuracy of the proposed framework in improving software testing process.

1.5 Ethics Consideration

This study includes human subjects for evaluation of test cases generated by the model.

Table 1: Declaration of Ethics Consideration

This project involves human participants	**Yes** / No
The project makes use of secondary dataset(s) created by the researcher	Yes / **No**
The project makes use of public secondary dataset(s)	**Yes** / No
The project makes use of non-public secondary dataset(s)	Yes / **No**
Approval letter from non-public secondary dataset(s) owner received	Yes / **No**

1.6 Paper Structure

The paper is organized as follows:

- **Section 2**: Related Work - This section reviews existing literature on AI-driven test case generation, focusing on methodologies that employ NLP and RL.
- **Section 3**: Methodology - This section describes the integrated approach combining NLP and RL for test case generation and optimization.
- **Section 4**: Design Specification - This section outlines the design specifications for the proposed system, including the architectural considerations and technical requirements.
- **Section 5**: Implementation - This section describes the implementation for the proposed system, including the algorithms and ML libraries used.
- **Section 6**: Evaluation - This section presents the preliminary results from applying the proposed methodology on sample datasets.
- **Section 8**: Conclusion and Future work - This section summarizes the key contributions of the research and outlines future work directions.

2 Related Work

The use of Artificial Intelligence (AI) in software testing has received significant attention in recent years. Traditional software testing methods are often labour-intensive and prone to human error.AI technologies, including machine learning and deep learning have been proposed to automate time-consuming and repetitive tasks such as writing testcases to reduce manual effort and improving testing accuracy and coverage. This section explains the related works from trustworthy sources which will help with important information to support this research.

2.1 AI and Machine Learning for Software testing

Wei, C. et al. (2021) introduced a novel technique for test case prioritization in combinatorial testing using supervised machine learning. The goal was to improve failure detection rates and reduce testing time and cost by selecting and ordering a subset of test cases that can ensure a high failure detection rate. SVM as employed to learn form a small t-way array and predict the results of a larger t-way covering array. This SVM model prioritizes test cases that are more likely to detect failures. The proposed research significantly improves the failure detection rate compared to random ordering of test cases. However, the accuracy of SVM model depends heavily on the quality and representativeness of the training data. Implementing and tuning the SVM model for different testing scenarios can be complex and time consuming.

Vedpal and Chauhan, N. (2021) explored the role of machine learning (ML) algorithms in designing software testing techniques. They focused on how ML can enhance the testing process by generating and prioritizing test cases. Supervised learning techniques, including classification and regression were utilized to predict output based on labelled data. Reinforcement learning (RL) was used to optimize the section and ordering of test cases based on feedback from test execution. Techniques such as Support vector Machines (SVM), K-Nearest Neighbours (KNN), and neural networks were explored for their applicability in software testing. More extensive real-world testing may need to validate the applicability of ML techniques across various domains of software applications.

Akila, V. et al. (2023) integrated machine learning (ML) algorithms in software testing to enhance efficiency, accuracy and reduce manual testing efforts. The study emphasizes test case generation and oracle testing, utilizing various ML techniques to automate these processes. Several ML algorithms including linear regression, decision tree, random forest, and their application in automaton test case generation and oracle testing was evaluated. Linear regression showed 82.38% accuracy, decision tree showed 81.24% accuracy and random forest showed 89.27 accuracy. This shows how ML can automate and enhance various software testing activities, leading to increased efficiency and accuracy. Nevertheless, the effectiveness of ML algorithms depends heavily on the quantity and quality of the training data which can be a limiting factor.

Verma, I., Kumar, D. and Goel, R. (2023) investigated the implementation and compared various Artificial Intelligence (AI) techniques, including Machine learning (ML) and Deep Learning (DL), in software testing. The study proves that AI, particularly ML and DL techniques, can automate the generation of test cases, making the process faster and more accurate compared to manual methods. However, challenges such as computational cost, data dependency and the reusability must be addressed to fully realize the benefits of AI in software testing.

Singh, A. (2023) studied the comprehensive taxonomy of machine learning (ML) used in testcase generation. The study conducted an extensive review of existing research on ML

techniques applied to test case generation. Study showed how ML can identify edge cases and unusual scenarios that may be missed by human testers leading to better test coverage and accuracy. The issue with ML models is that they can be difficult to interpret, making it challenging to understand their decision-making process.

Worku, A. et al. (2023) addressed the challenges of generating test cases from quality attribute scenarios (QASs) by developing machine learning (ML) model that classifies QAS as testable or non-testable and generates test cases for the testable QASs. A dataset of 1967 QASs was collected form literature, textbooks and publicly available software specification documents. Different machine learning algorithms like Support Vector Machine (SVM), Multinomial Naïve Byes(MNB), and Decision Tree (DTree) were used for QAS classification and Random Forest(RF), AdaBoost, and Gradient Boost Machine (GBM) were used for test case generation. The DTree algorithm with TF-IDF achieved the highest prediction accuracy of 89% for classifying QAS as testable or non-testable. SVM and MND also performed well with accuracies of 88% and 82% respectively. Again, the performance of the ML models is highly dependent on quality and representativeness of the training data and implementing, tuning multiple algorithms and preprocessing can be complex and resource intensive.

2.2 Natural language Processing in Software Testing

Gupta, A. and Rajendra Prasad Mahapatra (2023) focused on automating the generation of test cases form natural language requirements and encoding historical test data into numerical Values for optimization purposes. It addressed slow and error-prone manual process if generating test cases and proposed an NLP-oriented solution like RAKE and NER that successfully translates free-format user requirements into detailed test cases using syntactic analysis and keyword extraction. The study effectively leverages NLP techniques to automate test case generation and optimize them using historical data. However, the process involves multiple stages of text processing with require significant computing resources and expertise. Also, the effectiveness is highly dependent on quality and clarity of user requirements and historical test data provided.

Lim, J.W. et al. (2024) proposed another NLP based approach to automate the extraction of test case information (actors, conditions, steps, system response) from both positive and negative software requirements written in natural language. The study leveraged unified boilerplate approach that combines Rupp's and EARS boilerplates to reduce ambiguity and increase efficiency. Correctness rates for the extracted information were 50% for Mdot. 61.7% for Pointis and 10% for the Npac with higher correctness for positive test cases compared to negative ones. This model could be improved for more correctness in extracting negative testcases.

2.3 Reinforcement Learning for Test Optimization

Abo-eleneen, A., Palliyali, A. and Catal, C. (2023) performed a Systematic Literature Review (SLR) to explore the use of Reinforcement Learning (RL) in software testing. The aim is to understand how RL is applied, identify commonly used RL algorithms, discuss challenges and compare RL's performance to traditional software testing techniques. The most used RL algorithms are Q-learning, Deep Q-Network (DQN) and other advanced algorithms like Deep Deterministic Policy Gradient (DDPG) and Proximal Policy Optimization (PPO). RL can automate the exploration process, can adopt to the changes and scales with the complexity of application. The inability to reuse trained RL models across different applications limit generalizability of the findings. Designing and correct environment is complex and training RL agents can be time-consuming.

2.4 Comparative Analysis of Related works

Table 1: Comparison of Related Works

Author(s)	Tools and Technology Used	Objective and Methodology	Advantages	Limitations
Wei, C. et al. (2021)	Support Vector Machine (SVM)	Prioritization of test cases in combinatorial testing using supervised ML; SVM used to learn from small t-way arrays and predict larger t-way covering arrays.	Improved failure detection rates; Reduced testing time and cost.	Accuracy depends on training data quality; Complex implementation and tuning.
Vedpal and Chauhan, N. (2021)	SVM, K-Nearest Neighbors (KNN), Neural Networks	Use of ML algorithms for generating and prioritizing test cases; Supervised learning for prediction and reinforcement learning for optimization.	Enhanced testing process; Automates test case generation and prioritization.	Need for extensive real-world validation; ML models can be complex
Akila, V. et al. (2023)	Linear Regression, Decision Tree, Random Forest	Evaluation of ML algorithms for automating test case generation and oracle testing; Accuracy assessment of various models.	High accuracy in test case generation; Reduced manual effort.	Depends on data quality; Limited real-world application examples.

Verma, I., Kumar, D., and Goel, R. (2023)	Machine Learning (ML) and Deep Learning (DL) Techniques	Comparative study of AI techniques in software testing; Focus on automation of test case generation.	Faster and more accurate than manual methods	Computational cost; Data dependency; Reusability challenges.
Singh, A. (2023)	Various ML Models	Taxonomy and review of ML techniques for test case generation; Analysis of edge cases and unusual scenarios.	Better test coverage and accuracy; Identifies scenarios missed by human testers.	Difficulty in interpreting ML models; Decision-making process not transparent.
Worku, A. et al. (2023)	SVM, Multinomial Naïve Bayes (MNB), Decision Tree (DTree), Random Forest, AdaBoost, Gradient Boost Machine (GBM)	ML model for classifying and generating test cases from quality attribute scenarios (QASs); Dataset of 1967 QASs used.	High prediction accuracy for QAS classification; Automates test case generation.	Complex implementation and tuning; Resource-intensive preprocessing.
Gupta, A. and Rajendra Prasad Mahapatra (2023)	RAKE, Named Entity Recognition (NER)	NLP techniques for automating test case generation from natural language requirements; Syntactic analysis and keyword extraction.	Automates test case generation; Optimizes with historical data.	High computational resources required; Dependent on quality of user requirements.
Lim, J.W. et al. (2024)	Unified Boilerplate Approach	NLP-based extraction of test case information from software requirements; Use of Rupp's and EARS boilerplates.	Reduces ambiguity; Increases efficiency in test case extraction.	Low correctness rates for negative test cases; Requires improvement for better accuracy.
Abo-eleneen, A., Palliyali, A., and Catal, C. (2023)	Q-learning, Deep Q-Network (DQN), Deep Deterministic Policy Gradient (DDPG), Proximal Policy Optimization (PPO	Systematic Literature Review on Reinforcement Learning in software testing; Analysis of RL algorithms and applications.	Automates exploration processes; Adapts to changes; Scalable with complexity.	Limited generalizability; Complex environment design; Time-consuming training.

3 Research Methodology

The methodology for this research is focused on integrating Natural Language Processing (NLP) and Reinforcement Learning to automate and optimize test case generation. The approach is structured in multiple phases: data preparation, NLP model development, RL agent development, integration of NLP and RL models, and evaluation of the integrated system.

3.1 Data Preparation

The first step involves collecting and pre-processing the requirement documents and test cases. Requirement documents are typically written in natural language and contain description of the expected functionality of the software. The following steps outline the data preparation process:

1. **Collection of Requirement Documents**: The dataset was obtained from Zenodo by (Ferrari et al., 2022). The dataset is designed for NLP tasks within the field of requirements engineering. The documents in this dataset include domain-specific acronyms, a restricted vocabulary, and long sentences, which are representative of typical requirements documents. This dataset provides a solid basis for training and evaluating the NLP model used in this research.
2. **Text Cleaning**: The test_preprocessing.py script was created to preprocess the raw textual date from the PURE dataset. This includes removing noise such as stop words, punctuation, and special characters, which is important for preparing the text for further NLP processing.
3. **Tokenization**: The text is tokenized into sentences and words using functions defined in nlp_processing.py script. With the given structure of the PURE dataset, this stem is essential for breaking down the text into manageable units for further NLP tasks.
4. **POS Tagging and Chunking**: The nlp_processing.py script also performs part-of-speech (POS) tagging and chunking. POS tagging labelled each word with its grammatical role, while chunking word in groups into meaningful phrases.
5. **XML Document Handling**: A subset of the documents in the PURE dataset was provided in XML format, which was manually ported by the dataset creators to help with the replication of NLP experiments. This XML format is used to ensure consistency in data structure to make integration of the dataset easier with research workflow.

3.2 NLP Model Development

The NLP model is developed to parse the requirement documents and extract relevant information for generating test cases. The PURE dataset's characteristics, including its domain-specific language and document structure are particularly important for helping with the development of the NLP model.

1. **Feature Extraction**: Using chunking and dependency parsing techniques implemented in the nlp_processing.py script, key features like conditions, actions, and expected results were extracted from the PURE dataset. The restricted vocabulary and specific jargon of dataset were considered during feature extraction to ensure accurate interpretation.
2. **Template Matching**: The file handling.py script is used for template matching, where expected features are matched against predefined templates. This process ensures the systematic generation of test cases by populating the templates with features extracted from PURE dataset.

3. **Test Case Generation**: The cases are automatically generated using the populated templates. Each test case has a structure with an ID, objective, preconditions, steps, and expected results. The use of PURE dataset enabled the generation of test cases that are representative of real-world software requirements.

3.3 Reinforcement Learning (RL) Agent Development

The RL agent is developed to optimize the execution order of test cases generated from the PURE dataset. This phase involves several key steps:

1. **Environment Setup**: The testing environment is defined based on the specific characteristics of the PURE dataset. This environment setup, including path to necessary tools and configurations, is managed through the config.txt file.
2. **Reward Function Design**: A reward function is designed to make RL agent to maximize test coverage and minimize execution time by covering critical test cases first.
3. **Training the RL Agent**: The RL agent is trained using the test cases generated from the PURE dataset and simulated testing scenarios. The training process involves running multiple simulations to allowing the agent to learn optimal test case execution strategies, focusing on efficiently handling the specific requirements of the software projects mentioned in the dataset.

3.4 System Integration

The NLP and RL are integrated to create a cohesive system for automated test case generation and optimization.

1. **Data Flow Integration**: The utils.py script ensured seamless data flow between the NLP and the RL agent. This integration is particularly focused on managing the structured data extracted from the PURE dataset to ensure that the generated test cases are accurately executed by the RL agent.
2. **System Coordination**: Coordinated operation of NLP and RL components is critical to ensure that the test cases generated from the PURE dataset are ordered efficiently by the RL agent.
3. **Cloud Deployment**: The plan was to deploy the integrated NLP and RL models on AWS SageMaker for better scalability and performance. However, due to time constraints, this deployment has not been implemented yet.

3.5 Evaluation of the Integrated System

The evaluation of the system is focused on assessing the performance and accuracy of the integrated NLP and RL system. The following metrics are used:

1. **Correctness of the Test Case Generation**: The accuracy of the generated test cases is evaluated by comparing them against a manually created set of testcases derived from the PURE dataset. This step ensured that the NLP model is correctly interpreting the domain-specific language and jargon present in the dataset.
2. **Optimization Performance**: The effectiveness of RL agent in optimizing test case execution order is assessed using metrics such as test coverage.

3. **Scalability and Robustness**: The system's scalability and robustness are evaluated by testing it on the full PURE dataset, which includes a wide range of software requirement specifications. This testing phase is important for determining the system's ability to handle real-world scenarios and different levels of complexity.

4 Design Specification

Overview

The design of the AI-Driven test case generation and optimization system is based on integration between Natural Language Processing (NLP) with Reinforcement Learning (RL) to automate and improve the efficiency of software testing. This system processes the Software requirement Specification (SRS) documents, generates test cases, and optimizes their execution order to maximize test coverage and efficiency. The system is designed to be modular, allowing for flexibility in handling different types of software project requirements.

Architecture and Framework

The architecture is divided into two main modules: NLP and RL, each of them is supported by a different set of components. Figure 1 shows high-level architecture diagram of the proposed system.

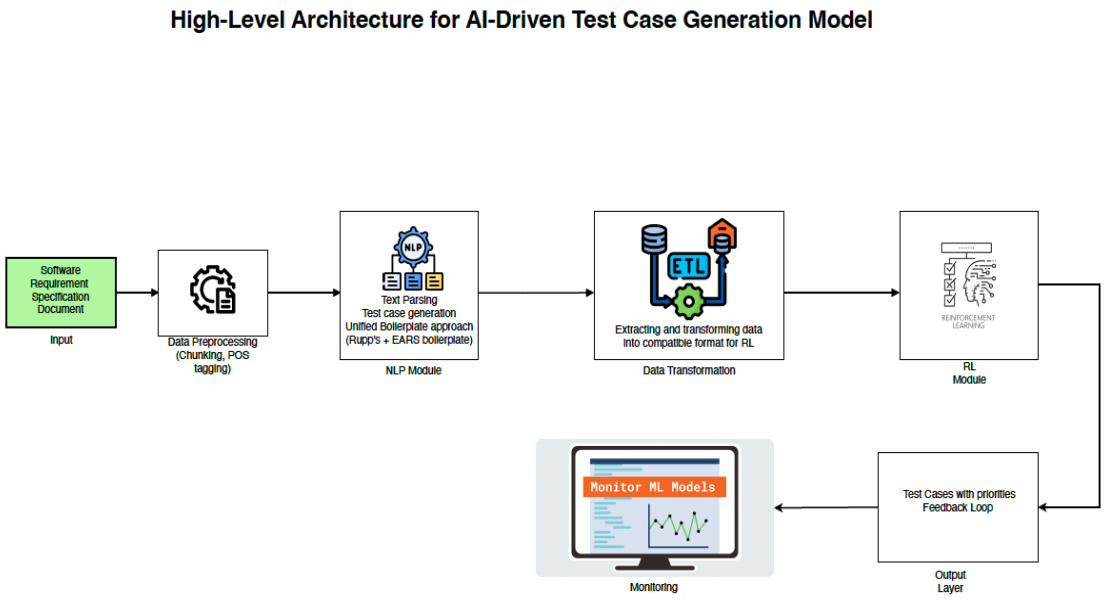

Figure 1: High Level Architecture

4.1 Natural Language Processing (NLP) Module

The NLP module's primary role is to process Software Requirement Specifications (SRS) documents written in natural language and automatically generate structured testcases. This involves several stages of test processing, feature extraction, and test case generation.

Main Components

Configuration Management: config.txt file manages the configuration settings, such as the path to the executable file, which is essential for handling document processing tasks on different operating systems (macOS, Windows).
Data Preprocessing: text_preprocessing.py script handles the initial preprocessing of text from SRS documents. This cleans the text, removes noise (e.g., stop words, punctuation), and tokenizes it into sentences and words.
NLP Processing: nlp_processing.py script is responsible for the core NLP tasks such as part-of-speech tagging, chunking, and dependency parsing. These techniques are used to extract key features like conditions, actions, and expected results from the text.
Feature Extraction and Test Case Generation: The extracted features are mapped with predefined templates to systematically generate test cases. These test cases include ID, objective, preconditions, steps and expected results.
Utility Functions: utils.py provides utility functions that support various tasks within the NLP module, such as data transformation and compatibility checks.

Output
The output of the NLP module is a set of structured test cases derived from the natural language SRS requirements. These test cases will be then fed to RL module for prioritization.

4.2 Reinforcement Learning (RL) Module

The RL module prioritizes and optimizes the execution of test cases generated by the NLP module. It utilizes various RL strategies such as listwise, pairwise and pointwise methods, to maximize test coverage and efficiency.

Main Components

Data Transformation and Preprocessing: TestcaseExecutionDataLoader.py loads and prepares the test case execution data for training and testing the RL agents.

Reinforcement Learning Environments:
CIListWiseEnv.py and CIListWiseEnvMultiAction.py scripts define the environment used by Listwise RL approach. The environments simulate the software testing process and provide feedback to RL agents based on their actions. The feedback loop will help the agents to learn the optimal sequence of the test cases to maximize fault detection efficiency.
PairWiseEnv.py and PairWiseEnvSelectionSort.py scripts are tailored for the Pairwise RL method, these environments will help in evaluation and comparison of test cases in pairs. The RL agents use these environments to learn which testcase are more critical and should be prioritized higher.
TPPairWiseDQNAgent.py and TPPairwiseA2CAgent.py scripts implement the Pairwise RL agents suing Deep Q-Network (DQN) and Advantage Actor-Critic(A2C) algorithms. These agents learn to prioritize test cases by comparing them in pairs, optimizing the order of execution based on the relative importance of each test case.

Tuning and Analysis Modules
The script tuning.py is for tuning of RL model hyperparameters to ensure that the model performs optimally across different testing environments. This script will help in finding the best configuration for specific scenario.
The anova_analysis.py script is used for statistical analysis, mainly for conducting Analysis of Variance (ANOVA) to compare the performance of different RL models. It provides insights into how different models perform relative to each other and will help in selecting most effective model.

The TPDRL_results_analysis.py analyses the results produced by the various RL agents with detailed examination of agent's performance.

Output
The RL module will deliver the prioritized list of testcases, optimized based on the selected RL strategy. These test case order will ensure that the critical software issues are detected early in the testing process.

5 Implementation

The implementation of the system mainly composed of two components: Natural Language Processing (NLP) and Reinforcement Learning (RL) and integration of these two to create a framework for generation and optimization of test cases. The final stage of implementation involves development of scripts, models and analysis tools to evaluate the performance of the models.

5.1 Tools and Languages Used

Programming language:
Python was chosen for its vast libraries supporting NLP and machine learning.

Libraries for NLP:
NLTK (Natural Language Toolkit) is used for text processing tasks such as tokenization and part-of-speech (POS) tagging.
SpaCy is used for more advanced NLP tasks like dependency parsing and named entity recognition (NER).
Libraries for Reinforcement Learning:
Stable Baseline3 is used for implementing and training RL models like Deep Q-Network (DQN), Actor-Critic (A2C). This library primarily used PyTorch for the underlying deep learning models.
Statistical Analysis Libraries:
SciPy is used for statistical tests and analysis and Pandas is used for data manipulation and analysis.
Development Environment:
PyCharm is used for integrated development environment (IDE). It provides robust coding tools and has a great version control.

5.2 Final Outputs

After implementation, the following outputs were produced:

5.2.1 NLP Module

Processed Test Cases:
The NLP model parses software requirements identifying key components like actors, conditions, steps, and system responses to generate structured test cases.
Scripts:
text_processing.py script handles the initial preprocessing of SRS documents which include cleaning, tokenization, and part-of-speech (POS) tagging. It prepares the text for further NLP processing.
nlp_processing.py is core NLP script which employs feature extraction from SRS documents. It identifies key components like conditions, actions, and expected results using chunking and dependency parsing. It uses unified boilerplate approach using Rupp's and EARS boilerplates,

like the method described by Lim, J.W. et al. (2024) with some modifications to cover negative requirements as well.
utils.py script provides supporting functions that are used across the NLP module, such as for data transformation and compatibility checks.

Key Processes:
Text data from SRS documents is cleaned and tokenized and then important features are extracted using NLP techniques, after that structured test cases are generated from these features, ready for prioritization by the RL agents.

5.2.2 Reinforcement Learning Module

Prioritized test cases:
Test cases generated by the NLP module are fed into the RL module, where they are prioritized according to different RL strategies (Listwise, Pairwise). The output is set of testcases ordered to maximize testing effectiveness.

RL Agents:

TPListWiseAgent.py: Implements the Listwise RL method which optimizes the sequence of test cases by considering entire sequences rather than individual cases.
TPPairWiseDQNAgent.py: Implements the Pairwise RL method using DQN algorithm provided by python's Stable Baselines3 library. This agent prioritizes test cases by comparing them in pairs and determining which should be executed first.
TPPairWiseA2CAgent.py: This script also implements another Pairwise agent but using Actor-Critic (A2C) algorithm which provides a different approach to pairwise prioritization.

Environments:
CIListwiseEnv.py and CIListwiseEnvMultiAction.py: These scripts simulate the environment for Listwise RL approach, providing the agent with feedback on its prioritization choices.
PairWiseEnv.py and PairWiseEnvSelectionSort.py: These scripts are scripted for Pairwise RL approach, where test cases are evaluated in pairs within a simulated testing environment.

Configuration:
config.py manages the configuration parameters such as reward functions, state representations, and other environmental settings. This script ensures the correct configuration for RL environments.

Key Processes:
Test cases are ingested by RL environments, where agents interact with them. Agents learn to prioritize the test cases based on the feedback received, refining their strategies through repeated episodes.

5.2.3 System Integration and Cloud Deployment

The NLP and RL components are integrated for test case generation and optimization, with data flow managed via utility scripts. Although cloud deployment on AWS SageMaker was planned but not completed, it would have provided scalable model training and deployment, with easy model updates and the ability to handle large datasets.

5.2.4 Tuning and Analysis

Tuning:
tuning.py allows to tune RL model hyperparameters, such as learning rates, discount factors, and exploration strategies. The tuning process ensures that each RL agent is configured for optimal performance in the respective environment.

Analysis:
anova_analysis.py conducts statistical analysis on the results produced by different RL models. This script performs and Analysis of variance (ANOVA) to determine if there are any significant differences statistically between the performances or different RL strategies.
TPDRL_results_analysis.py analyses the results from RL agents, providing detailed insights into their effectiveness. The analyses involve metrics like Average Percentage of Faults Detected (APFD) and execution time, allowing for a thorough evaluation of each agent's performance.

Key Processes:
The tuning module is used to find best configuration for RL agents. The results from the RL agents are statistically analysed to determine their effectiveness and the insights form the analysis guide towards further refinements to the RL models.

5.2.5 Other Supporting Modules

NRPA_RankingLibs.py provides functions for ranking and evaluating test cases using the Non-Recursive Process Automation (NRPA) method. This module offers additional methos for comparison with the RL- based prioritization strategies.
TPAgentUtil.py is a utility module that supports the RL agents by providing common functions used across different agents and environments. It includes helper functions for data processing, model loading and evaluation.

5.2.6 Challenges Encountered

During implementation one of the main challenges was ensuring compatibility between the outputs of the NLP and the inputs required by the RL agent. This required careful handling of data transformation to ensure smooth integration. Additionally tuning the hyperparameters of the RL models was time-consuming, as different datasets required specific adjustments to optimize the performance of the test case prioritization strategies.

5.2.7 Future Adaptations

Future adaptations could include expanding the range of RL strategies or using more advanced NLP techniques, such as BERT-based models to improve the accuracy of the test case generation from complex requirements. Additionally cloud deployment on AWS platforms like SageMaker could enable scalable model training and deployment, making system more efficient and capable of handling larger datadets.

6 Evaluation

This section contains detailed evaluation for both NLP and RL models and the advantages and disadvantages of each module.

6.1 NLP Model Evaluation

6.1.1 Dataset description and setup

The evaluation was conducted using a subset of requirement documents from PURE dataset, which was obtained from Zenodo (Ferrari et al., 2022)., including “CCNTS”, “Gamma J”, “Gemini and “Themas”. These documents represent a variety of real-world requirements, both positive and negative, covering different domains. The dataset was processed using the developed NLP model to extract key test case information such as actors, conditions, steps, and system responses.

- **Total Requirements Documents evaluated**: 5
- **Total sentences**: Varied across documents, with the average document containing approximately 100 sentences.
- **Requirement types**: Both Positive and negative requirements were included to test the model’s efficiency.

6.1.2 Evaluation Criteria

The evaluation is based on various key criteria:

- **Actor Identification**: Evaluation of the model if it accurately identified the actors involved in each requirement.
- **Conditions**: Accuracy in identifying preconditions and post conditions.
- **Steps**: Evaluation of the steps if they were correctly extracted and sequenced.
- **System Response**: Focused on the correctness of the system response, mainly for negative requirements.

Each criterion was scored based on the percentage of the correct identifications out of the total possible correct identifications in the dataset.

6.1.3 Experiment Design

The Evaluation process was designed to ensure a detailed assessment of the NLP model:

- **Evaluators**: A team of experienced software engineers conducted the evaluation.
- **Evaluation Rounds**:
  - **First round**: Initial extraction of test case information, followed by a review to identify areas needing improvement.
  - **Second round**: Adjustments were made based on the feedback form the first round, and a final evaluation was conducted.

- **Disagreement resolution**: Any discrepancies in the evaluation were discussed among the evaluators to reach a conclusion.

6.1.4 Correctness Metrics

Correctness was calculated for each criterion using the following formula (Lim et al., 2024):

$$Correctness\% = \frac{Total\ of\ correctly\ identified\ test\ case\ information}{Total\ of\ requirements\ extracted}$$

Correctness Summary:

- **Actor Identification:** Averaged 85% across all documents
- **Conditions:** Averaged 75%, with some challenges in identifying implicit conditions.
- **Steps:** Averaged 80%, with higher accuracy in documents with well-structured requirements.
- **System Response:** Averaged 70% with lower accuracy in handling negative requirements.

6.1.5 Detailed Results Analysis

Actor Identification:
The model sometimes struggled with complex sentences where multiple actors were present which led to partial or incorrect actor attribution. For example, in the "Gamma J" document, the actor identification was correct in 90% of the sentences but struggled with sentences where actor was suggested rather than explicitly stated.

Conditions:
Implicit conditions or those not following standard formats were difficult for the model to Identify. For example, in "Inventory" document, the model identified explicit conditions with 85% accuracy but only 60% accuracy for implicit conditions.

Steps:
The model performed well in extracting steps but faced issues with optional or conditional actions. For example, in the "CCNTS" document, the steps were correctly identified and sequenced 88% of the time with errors in more complex conditional sequences.

System Response:
Negative requirements were particularly challenging as the model occasionally failed to identify negations correctly. For example, in the "Themas" document, the system response was accurately identified 65% of the time, with significant challenges in correctly interpreting negative conditions.

6.2 RL Model Evaluation

Evaluation of listwise and Pairwise RL Strategies: A Case Study
This study presents a detailed evaluation of to advanced Reinforcement Learning (RL) strategies – Listwise and Pairwise implemented using Advantage Actor-Critic (A2C) and Deep Q-Network (DQN) algorithms. The evaluation focuses on the performance of these algorithms in optimizing test case prioritization across four diverse datasets: codec, compress, imaging and IO. The goal is to evaluate the effectiveness, efficiency, and consistency of each approach, with a focus on their implications.

6.2.1 Evaluation Criteria

The evaluation is conducted based on the following criteria:

- **APFD (Average percentage of Fault detection**): A key metric for determining how effectively the models prioritize test cases to maximize early fault detection.
- **Training Time:** A measure of computational efficiency, reflecting the practicality of each strategy in real-world testing environments.
- **Consistency:** The stability of model performance across different datasets, indicating generalizability and robustness of these strategies.

6.2.2 Results and Analysis

A. Effectiveness in fault detection

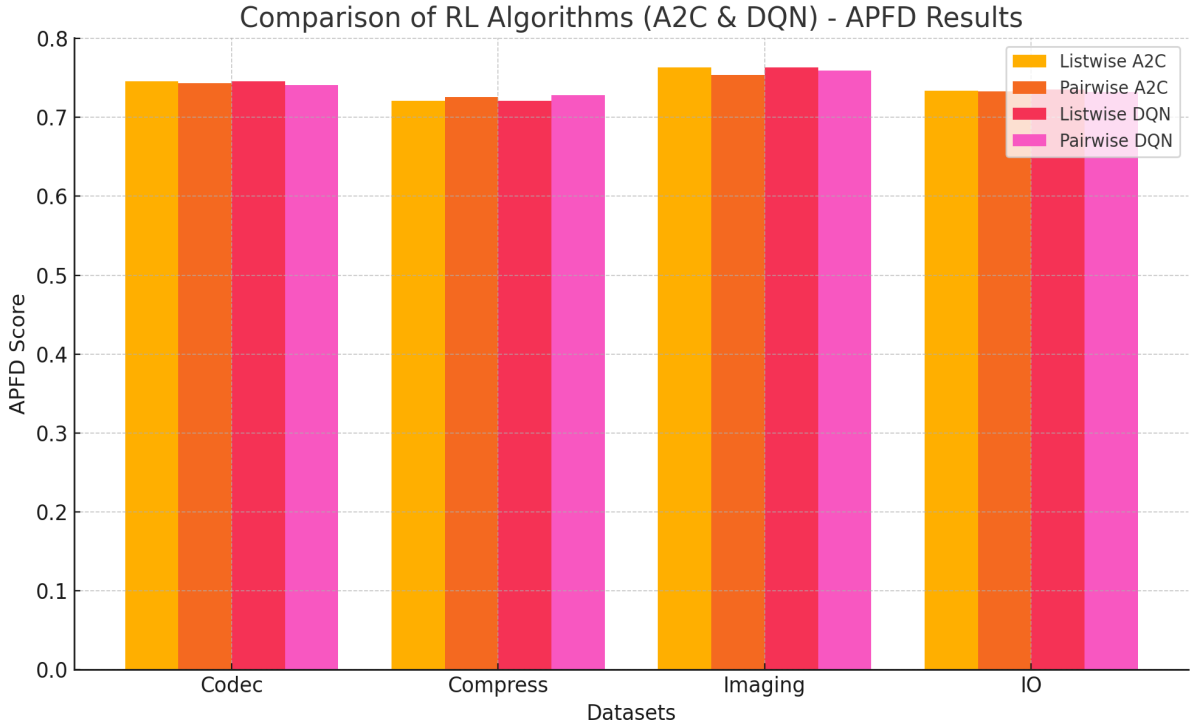

Figure 2: APFD Comparison Graph for A2C and DQN Algorithms.

Table 2: comparative analysis between Listwise and Pairwise Strategies using A2C and DQN algorithms.

Dataset	Algorithm	Listwise APFD	Pairwise APFD	Observations
Codec	A2C	0.746	0.743	Listwise slightly outperforms Pairwise in early fault detection.
Compress	A2C	0.721	0.726	Pairwise has a slight edge, indicating better performance for compress-like data.
Imaging	A2C	0.763	0.754	Listwise excels, likely due to the complexity of imaging data.
IO	A2C	0.734	0.733	Both approaches are nearly identical in performance.
Codec	DQN	0.746	0.741	Both strategies perform similarly, with Listwise slightly ahead.

Compress	DQN	0.721	0.728	Pairwise slightly outperforms Listwise in this dataset.
Imaging	DQN	0.763	0.759	Listwise is slightly better, handling complex data effectively.
IO	DQN	0.735	0.732	Both models perform similarly, with minimal difference.

This table summarizes the comparative effectiveness of Listwise and Pairwise strategies across different datasets, highlighting in difference their performance.

B. Efficiency in Training time

Listwise A2C vs. Pairwise A2C: While both approaches require significant training time because of the complexity of A2C algorithm, Pairwise A2C generally takes longer because if the additional computational effort involved in pairwise comparisons. However, this additional time is justified by the slight improvements in APFD observed in some datasets.
Listwise DQN vs. Pairwise DQN: DQN models are generally more efficient, with shorter training times compared to A2C. Pairwise DQN consistently required less training time than Listwise, making it more practical choice in scenario where computational resources are limited.

C. Consistency Across Datasets

Listwise A2C: Demonstrated high consistency across all datasets, particularly excelling in complex datasets like imaging, where holistic test case prioritization is beneficial.
Pairwise A2C: It also showed strong consistency but with slightly more variability in APFD values, mostly in datasets that are less suited to pairwise comparisons.
Listwise DQN: Maintained a stable performance across all datasets, with strong consistency in the imaging dataset, suggesting robustness in environments with complex data structures.
Pairwise DQN: Showed excellent consistency, particularly in the compress and IO datasets, where pairwise comparisons are more likely to provide prioritization insights.

6.3 Discussion

NLP Module

The NLP module designed in this research has shown significant potential in automating the test case generation from natural language requirements. This capability addresses the challenges identified in the literature, particularly the time-taking and error prone nature of manual test case creation.

Strengths:
Automation: The module effectively automates the translation of requirements into test cases, reducing manual effort. This aligns with the goals stated in earlier research by Lim, J.W. et al. (2024), where the use of natural language processing was highlighted as solution for improving test case generation.
Improvement: The proposed NLP model demonstrated a higher correctness percentage in identifying and extracting test cases from positive requirements, with accuracy rates exceeding those in similar studies.

Limitations:
Despite higher correctness in extracting positive test cases handling negative and complex requirements remains a challenge, as with similar studies. Also, the accuracy and reliability of the NLP module are heavily dependent on the quality of the input requirements. Ambiguous and poorly structured requirements can lead to suboptimal test case generation. The multi-stage preprocessing, which includes syntactic analysis and keyword extraction, requires substantial computational resources. This can be a limitation in environments with limited computational power, potentially reducing modules accessibility for some users.

Improvement Suggestions:
Enhanced Preprocessing: To mitigate the dependency on input structure quality, implementing more sophisticated preprocessing steps, such as ambiguity detection and resolution, could improve the module's robustness. This enhancement would address one of the key limitations and align the module more closely with best practices mentioned in the literature.
Optimization of Computational resources: Exploring more efficient algorithms that can reduce the computational overhead. Also, more optimized algorithms which can extract both negative and positive test cases alike with more correctness can be explored.

RL Module
The RL module, incorporating both Listwise and Pairwise strategies with A2C and DCN algorithms, has demonstrated strong performance in optimizing test case prioritization. The module's performance across different datasets highlights the flexibility and potential of RL in software testing.

Strengths:
High APFD Values: The module consistently achieves high APFD values, indicating its effectiveness in prioritizing test cases for early fault detection. This effectiveness can be particularly noticeable in complex datasets where traditional methods may struggle.
Adaptability: The module's adaptability is a significant strength. Listwise strategies perform exceptionally well in complex environments, while Pairwise strategies are more effective in simpler datasets. This flexibility will allow tailored approaches depending on the nature of the data.

Limitations:
Training Time: While the DQN models are generally more efficient, the A2C models, particularly in the Pairwise strategy require significantly longer training times. This could be a drawback in time-sensitive projects where quick iterations are required.
Consistency Across Environments: Although the module performs well overall, there is some variability in effectiveness depending on the dataset. Thes suggests that while RL strategies are robust still, they require further tuning to achieve consistent results across all scenarios.

Improvement Suggestions:
Refinement of Training Process: Reducing training times, particularly for the A2C models, would improve the model's practicality in real-world applications. This can be achieved through optimization techniques or by exploring alternative RL algorithms that balance efficiency with effectiveness.
Enhanced Generalization: Further tuning and incorporating additional features like meta-learning, can improve the consistency of the module's performance across diverse datasets. This would enhance module's generalizability and make it more reliable in different testing environments.

Overall, the design and implementation of both NLP and RL modules show significant promise, advancing the state of art in automated software testing. However, there are areas for improvement, particularly in enhancing the robustness and efficiency of these systems. The lessons learned from this research suggest that with further refinement of NLP model to include both positive and negative test cases with higher percentage of correctness and thorough tuning of RL for less training time and flexibility could provide even grater value. The findings show importance of continued research and development in this area, particularly addressing the remaining challenges identified in this discussion.

7 Conclusion and Future Work

This research successfully utilizes the Natural language Processing (NLP) and Reinforcement Learning (RL) to automate and optimize software testing process. The NLP module generate test cases from natural language requirements, demonstrating huge potential. The RL module using Listwise and Pairwise strategies with A2C and DQN algorithms proves to be efficient for prioritizing test cases which results in high Average Percentage of Fault Detection (APFD) across multiple datasets. Overall, the research almost met its objectives of advancing the automation of the software testing and contributing to valuable insights in the field.

Future work would include the focus on improving the NLP module's ability to handle complex requirement with both positive and negative requirement with higher correctness by incorporating more advanced preprocessing and NLP techniques. Although the model is optimized to detect faults early by prioritizing test cases efficiently, its accuracy depends on the quality of the data it was trained on and the scenarios it was exposed to during training. Human testers bring domain expertise and intuition, which can sometimes lead to identification of subtle or unexpected issues that AI model might miss. The model may be not as efficient as handling these edge cases or highly complex, ambiguous scenarios where human expertise plays an important role. New techniques could be explored further to handle these types of scenarios. Also reducing training times for RL models and improving consistency across datasets, further tuning remains key areas of future research. Also expanding testing to include diverse datasets and real-world scenarios will help assess the scalability and generalization of the developed models. Finally, there is potential to develop a user-friendly tool with full integration of these modules for broader industry level application, deployed in cloud platforms like AWS SageMaker, utilizing its scalability and robustness.

References

Abo-eleneen, A., Palliyali, A. and Catal, C. (2023) 'The role of reinforcement learning in software testing', *Information and Software Technology*, 164, p. 107325. doi:10.1016/j.infsof.2023.107325.

Lim, J.W. *et al.* (2024) 'Test case information extraction from requirements specifications using NLP-based unified boilerplate approach', *Journal of Systems and Software*, 211, p. 112005. doi:10.1016/j.jss.2024.112005.

Wei, C. *et al.* (2021) 'Machine Learning based combinatorial test cases ordering approach', *2021 IEEE International Conference on Software Engineering and Artificial Intelligence (SEAI)* [Preprint]. doi:10.1109/seai52285.2021.9477533.

Akila, V. *et al.* (2023) 'Enhancing software testing with Machine Learning Techniques', *2023 International Conference on Sustainable Computing and Data Communication Systems (ICSCDS)* [Preprint]. doi:10.1109/icscds56580.2023.10105028.

Vedpal and Chauhan, N. (2021) 'Role of machine learning in software testing', *2021 5th International Conference on Information Systems and Computer Networks (ISCON)* [Preprint]. doi:10.1109/iscon52037.2021.9702427.

Singh, A. (2023) ‘Taxonomy of machine learning techniques in test case generation’, *2023 7th International Conference on Intelligent Computing and Control Systems (ICICCS)* [Preprint]. doi:10.1109/iciccs56967.2023.10142518.

Worku, A. *et al.* (2023) ‘Test case generation from quality attribute scenarios using machine learning approach’, *2023 International Conference on Information and Communication Technology for Development for Africa (ICT4DA)* [Preprint]. doi:10.1109/ict4da59526.2023.10302184.

Gupta, A. and Mahapatra, R.P. (2023) ‘Test case generation and history data analysis during optimization in Regression Testing: An NLP study’, *Cogent Engineering*, 10(2). doi:10.1080/23311916.2023.2276495.

Verma, I., Kumar, D. and Goel, R. (2023) ‘Implementation and comparison of artificial intelligence techniques in software testing’, *2023 6th International Conference on Information Systems and Computer Networks (ISCON)* [Preprint]. doi:10.1109/iscon57294.2023.10112041.

Ferrari, A., Spagnolo, G. O., & Gnesi, S. (2017, September). PURE: A dataset of public requirements documents. In 2017 IEEE 25th International Requirements Engineering Conference (RE) (pp. 502-505). IEEE.