Report on

Test Case Generation Using Retrieval-Augmented Generation (RAG) and Reinforcement Learning with Human Feedback (RLHF)

Submitted in partial fulfillment of the requirements for the award of the

Bachelor of Technology

in

Computer Science and Business Systems

From

GOKARAJU RANGARAJU INSTITUTE OF ENGINEERING AND TECHNOLOGY

by

SHANKARAMPETA RITHVIK REDDY

Under the Esteemed guidance of

Shri.Vaddi Chandra Shekar
Scientist 'C' DRDL

Defence Research & Development Laboratory

Defence Research & Development Organisation

Kanchanbagh, Hyderabad, 500058.

ACKNOWLEDGEMENT

I state my sincere thanks to Shri G. A. Srinivas Murthy, Director, DRDL, Defence Research & Development Organization, and for providing me with this excellent opportunity to gain work experience in this highly esteemed organization and obtain knowledge in LLMs and Neural Network.

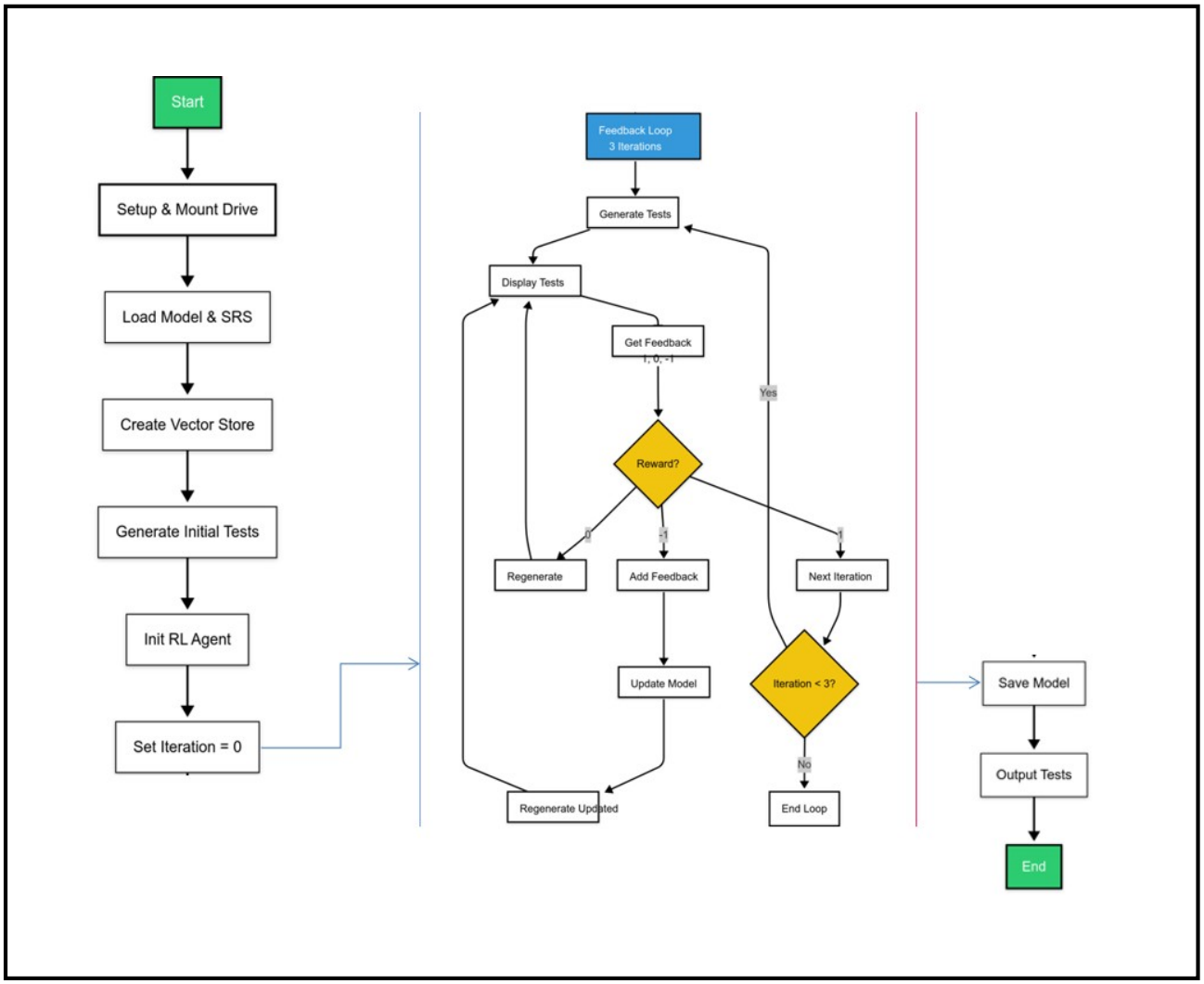
I sincerely convey my thanks to Shri. A Naresh , Scientist 'F', Shri. Vaddi Chandra Shekar , Scientist 'C' for giving me an opportunity to be an intern and for their valuable guidance in this field. I wish to express my gratitude to Shri.Vaddi Chandra Shekar , Scientist 'C' , Shri. Boda Nehru, Scientist 'D' DRDL for their invaluable teaching in this element.

Introduction

In the realm of software development, ensuring the reliability and functionality of applications is paramount. One critical aspect of this assurance is the generation of comprehensive test cases derived from software requirements. Traditional methods of test case generation often involve manual processes that are time-consuming and prone to human error. With the advent of advanced machine learning techniques, there is a growing interest in automating this process to enhance efficiency and accuracy.

This report delves into the integration of Retrieval-Augmented Generation (RAG) and Reinforcement Learning with Human Feedback (RLHF) to automate the generation of test cases from Software Requirements Specifications (SRS). By leveraging these techniques, we aim to create a system that not only understands the context of the requirements but also learns and adapts based on human feedback to produce high-quality test cases.

Architecture Diagram



Retrieval-Augmented Generation (RAG)

Overview of RAG

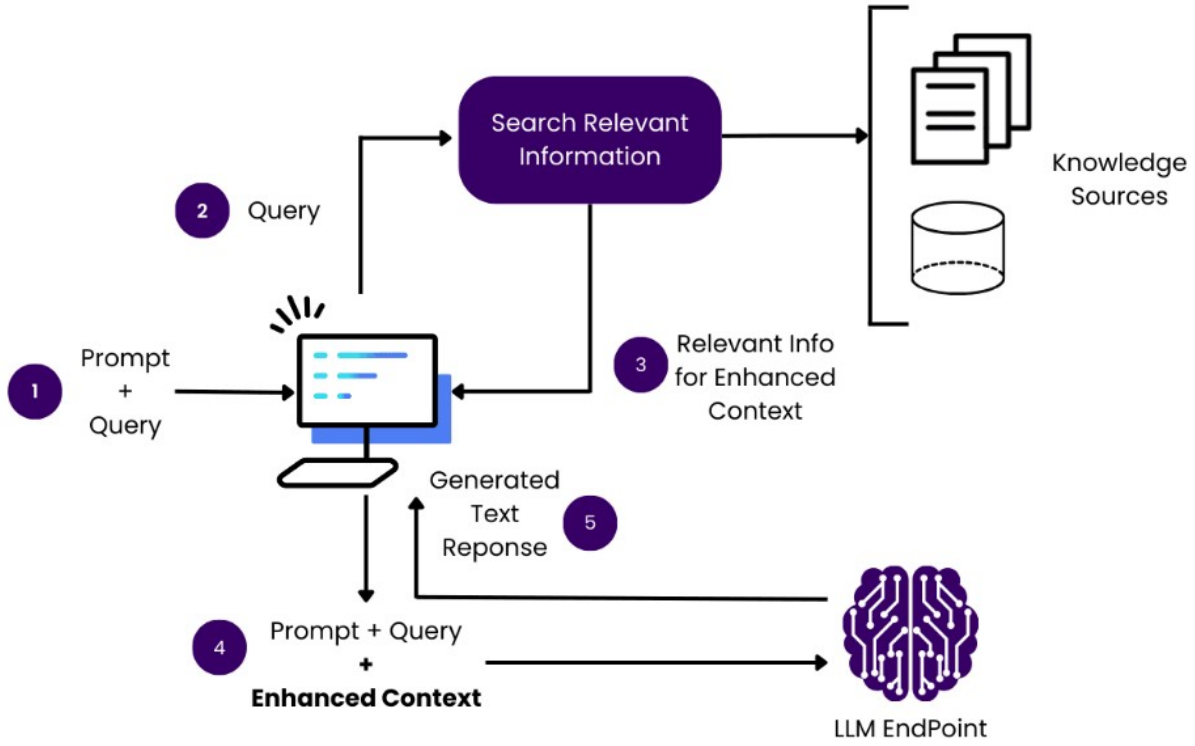
Retrieval-Augmented Generation (RAG) is a hybrid approach that combines the strengths of information retrieval systems with generative models. In traditional generative models, the system relies solely on the data it was trained on, which can lead to outdated or irrelevant responses. RAG addresses this limitation by incorporating an external knowledge base, allowing the model to retrieve relevant information dynamically and generate responses grounded in current and specific data.

The RAG architecture typically involves two main components: a retriever and a generator. The retriever searches a vast corpus of documents to find passages relevant to the input query. These retrieved passages are then fed into the generator, which produces a response that integrates the information from the retrieved documents. This mechanism ensures that the generated output is both contextually relevant and factually accurate.

In the context of test case generation, RAG can be employed to retrieve pertinent sections from the SRS that relate to a specific functionality or requirement. By doing so, the generative model can produce test cases that are directly aligned with the documented requirements, ensuring comprehensive coverage and relevance.

Moreover, RAG's ability to access external knowledge bases makes it particularly useful in scenarios where the model

needs to incorporate domain-specific information that may not be present in its training data. This feature is crucial for generating test cases in specialized fields such as healthcare, finance, or aerospace, where understanding domain-specific requirements is essential.



Benefits of RAG

One of the primary advantages of RAG is its ability to produce more accurate and contextually relevant responses compared to traditional generative models. By grounding the generation process in retrieved documents, RAG reduces the likelihood of producing hallucinated or incorrect information, a common issue in models that rely solely on their training data.

Another significant benefit is RAG's adaptability. Since the retriever component can access up-to-date information from

external sources, the system can provide responses that reflect the most current data, making it highly suitable for applications that require real-time information, such as news summarization or customer support.

In the realm of test case generation, RAG's ability to retrieve specific sections of the SRS ensures that the generated test cases are directly tied to the documented requirements. This alignment enhances the relevance and effectiveness of the test cases, leading to more thorough testing and higher software quality.

Furthermore, RAG's modular architecture allows for flexibility in its components. Developers can choose different retrieval mechanisms or generative models based on the specific needs of their application, enabling customization and optimization for various use cases.

Challenges and Considerations

Despite its advantages, implementing RAG comes with certain challenges. One of the primary concerns is the quality of the retrieval process. If the retriever fails to fetch relevant documents or retrieves misleading information, the generator's output may be compromised. Ensuring the retriever's accuracy is therefore critical to the overall performance of the RAG system.

Another consideration is the computational complexity. Integrating retrieval and generation processes can increase the system's latency, which may be problematic for applications

requiring real-time responses. Optimizing the retrieval and generation pipelines to minimize latency is essential for maintaining system performance.

Additionally, maintaining and updating the external knowledge base used by the retriever is crucial. The relevance and accuracy of the retrieved documents depend on the quality of the knowledge base. Regular updates and curation are necessary to ensure that the system continues to provide accurate and current information.

Lastly, integrating RAG into existing systems may require significant architectural changes, especially if the current infrastructure is not designed to support dynamic retrieval and generation processes. Careful planning and resource allocation are necessary to facilitate a smooth integration.

Applications of RAG

RAG has found applications across various domains due to its ability to generate contextually rich and accurate responses. In customer support, RAG-powered chatbots can retrieve and present information from a company's knowledge base, providing users with precise answers to their queries. This capability enhances customer satisfaction and reduces the workload on human support agents.

In the field of education, RAG can be used to develop intelligent tutoring systems that provide students with explanations and resources tailored to their specific questions.

By accessing a vast repository of educational materials, these systems can offer personalized learning experiences.

In healthcare, RAG can assist medical professionals by retrieving and summarizing relevant medical literature, aiding in diagnosis and treatment planning. The ability to access and synthesize up-to-date medical information is invaluable in a field where knowledge is constantly evolving.

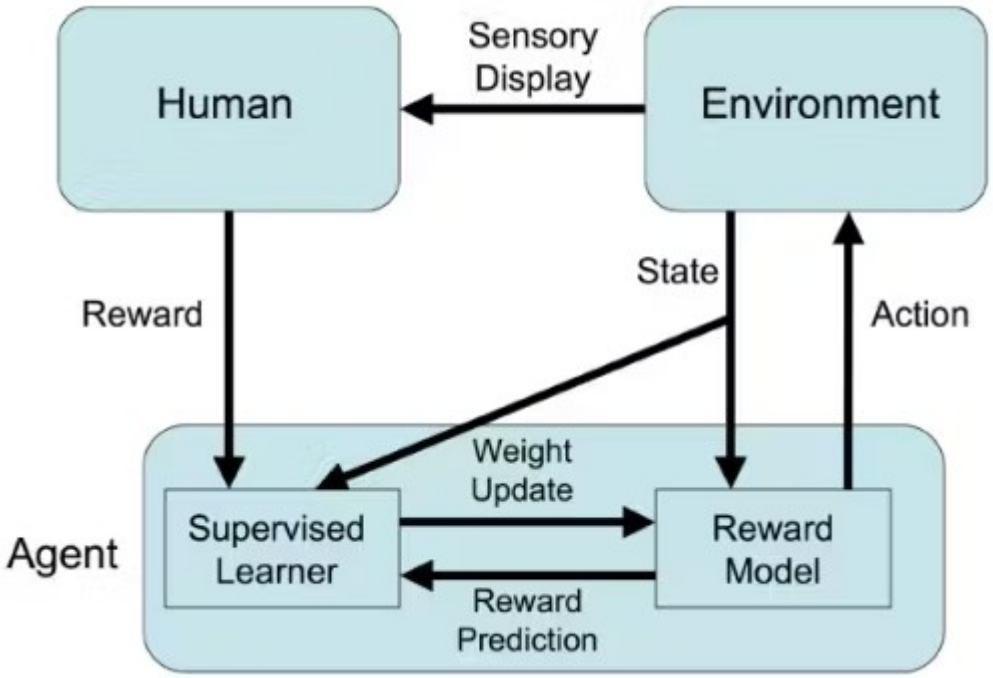
For software development, as explored in this report, RAG can automate the generation of test cases by retrieving relevant sections from the SRS. This application not only accelerates the testing process but also ensures that the test cases are comprehensive and aligned with the documented requirements.

Reinforcement Learning with Human Feedback (RLHF)

Overview of RLHF

Reinforcement Learning with Human Feedback (RLHF) is a technique that combines reinforcement learning with human-provided feedback to train models that align more closely with human preferences and values. Traditional reinforcement learning relies on predefined reward functions to guide the learning process. However, defining appropriate reward functions for complex tasks can be challenging.

RLHF addresses this issue by incorporating human feedback into the training loop. Instead of relying solely on predefined rewards, the model receives evaluations from human annotators



1. Introduction to Reinforcement Learning with Human Feedback (RLHF)

Reinforcement Learning with Human Feedback (RLHF) is a powerful framework that enhances machine learning models by integrating human input into the training process. In this project, RLHF was employed as an extension of the Retrieval-Augmented Generation (RAG) approach to refine the quality of test cases generated from a software requirements specification (SRS) document. While RAG allows the model to use relevant context retrieved from the document to generate responses, RLHF further improves these outputs through iterative learning guided by user feedback.

The RLHF framework allows for the incorporation of qualitative judgments from human evaluators, transforming subjective opinions into actionable signals that guide model optimization. In the context of test case generation, this feedback loop ensures that the generated test cases are not only relevant but also align with the specific quality, coverage, and formatting expectations set by human testers or project stakeholders. This dual-stage architecture ensures continual improvement over time.

By integrating RLHF, the model can overcome limitations associated with purely automated generation techniques, which often struggle to meet nuanced expectations in real-world software testing scenarios. The feedback loop supports continuous improvement and provides a mechanism for aligning model outputs with domain-specific knowledge that may not be encoded in training data. This process makes

RLHF a compelling choice for high-quality, context-sensitive applications.

In summary, RLHF plays a pivotal role in refining the test case generation model by embedding human insight into the training loop. This approach ultimately leads to more accurate, meaningful, and useful test cases, helping bridge the gap between AI-generated suggestions and real-world applicability in software testing environments.

2. Implementation of RLHF in the Test Case Generation Pipeline

The RLHF component was integrated into the existing RAG-based test case generation system. The process began with using a fine-tuned language model and a semantic embedding-based retriever that utilizes FAISS and HuggingFace embeddings. After generating initial test cases from the retrieved context of the SRS document, a human evaluator provides feedback on each generated output. The model is then fine-tuned in a lightweight manner using reinforcement learning principles.

A custom RL agent class was defined to manage feedback collection and model updates. For every generated test case, the user provides a numerical reward: 1 for good responses, 0 for neutral, and -1 for unsatisfactory ones. This reward modifies the model's behavior by adjusting the loss function during fine-tuning. Specifically, the model undergoes backpropagation with an adjusted loss computed as the

negative product of the reward and the model's prediction loss. This technique effectively promotes beneficial behaviors while penalizing poor ones.

To ensure consistent feedback and diversity in training, the system iteratively cycles through ten types of test cases such as functional, edge, performance, usability, and more. After each iteration, human feedback is gathered and the model parameters are updated accordingly. This continuous training cycle enables the model to adapt to a wide variety of test case categories and understand subtle variations in requirements interpretation.

The entire feedback loop is implemented with memory management precautions to prevent GPU memory overflows. Techniques such as gradient clearing and CUDA cache cleaning are integrated into the loop to ensure system stability. At the end of the cycle, the fine-tuned model is saved for future reuse. This complete integration of RLHF into the pipeline significantly enhances the precision and relevance of generated test cases.

3. Human Feedback Integration and Reward Mechanism

Human feedback acts as a cornerstone of the RLHF process. In this system, user feedback is collected in a structured yet flexible manner. Each output generated by the model is displayed to the user, who evaluates it based on criteria such as relevance, clarity, coverage, and adherence to requirement specifications. The feedback is numerical, ensuring that it can be easily integrated into the reinforcement learning loop.

This reward mechanism serves as the bridge between human qualitative judgment and machine learning optimization. The scale used is simple—1 for satisfactory results, 0 for average, and -1 for poor results—yet effective. This simplicity helps maintain a consistent feedback mechanism while reducing the cognitive load on the human evaluator, enabling more efficient and repeated interactions over multiple iterations.

The design of this mechanism encourages the model to learn not only from positive reinforcements but also from constructive criticism. When a user gives a -1 score, the system allows them to provide additional qualitative input explaining the deficiencies. This additional context becomes part of the prompt for the next iteration, creating a feedback-aware generation process that evolves over time.

Over multiple iterations, this feedback loop transforms the model's generation behavior. As it receives more granular and task-specific evaluations, the model fine-tunes its internal representation of what constitutes a high-quality test case. This learning process mimics human iterative learning and results in significant improvements in contextual alignment and functional adequacy of generated outputs.

4. Multi-Round Iterative Training with Diverse Test Case Types

To ensure comprehensive learning and generalization, the RLHF loop incorporates ten distinct types of test cases. These include functional, edge, performance, usability, security, compatibility, regression, integration, negative, and

exploratory test cases. Each type serves a different purpose in the software testing lifecycle and brings its own unique requirements and complexity.

In each iteration, the system generates test cases specific to one of these categories using the same underlying SRS context. This approach ensures that the model learns to adapt its response style and content based on the expected format and objectives of different test case types. As a result, the model becomes versatile and capable of producing a wide variety of test cases aligned with industry-standard testing practices.

The diversity of test case types also challenges the model to broaden its understanding of software requirements. For example, generating security test cases may require the model to reason about potential threats, while usability test cases focus on user experience. This breadth of exposure strengthens the model’s generalization ability and improves its performance across various testing domains.

Furthermore, by exposing the model to different evaluation criteria in each round, the RLHF loop ensures well-rounded development. The human feedback loop captures the strengths and weaknesses of the model in each area, helping guide the model toward more effective and adaptable test case generation. This structured iteration promotes a holistic enhancement of the model's capabilities.

5. Model Update Strategy

In Reinforcement Learning with Human Feedback (RLHF), updating the model based on user feedback plays a crucial role in fine-tuning it for higher accuracy and relevance. In the presented implementation, after generating test cases, the system prompts the user for feedback in the form of a scalar reward: 1 for good, 0 for neutral, and -1 for bad. This reward becomes the basis for updating the model's parameters using gradient descent. The reward is multiplied with the computed loss (cross-entropy) from the language model to scale the influence of each training instance proportionally to its quality as judged by the human reviewer.

The update mechanism is lightweight but effective for controlled fine-tuning. Using a smaller learning rate (1e-5), the optimizer (Adam) ensures that each update is subtle enough to avoid catastrophic forgetting of previously learned information. The gradients of the adjusted loss are computed and used to backpropagate through the model, gradually improving its performance in generating test cases aligned with user expectations. By treating the feedback as a reinforcement signal, the system learns to associate certain generation styles and information patterns with higher rewards.

An important nuance in the model update strategy is the token-level update tied directly to the prompt and model output. Instead of fine-tuning the model from scratch on a new dataset, this selective method allows the model to learn from minimal data samples interactively. This is a practical

advantage when computational resources or time are limited. It promotes adaptability without retraining on large datasets, which would otherwise be required in traditional supervised fine-tuning scenarios.

In the broader context of reinforcement learning, this strategy mimics reward-modulated supervised learning, blending classic language modeling with human-in-the-loop systems. Over multiple iterations, the model gradually becomes better at producing refined and high-quality test cases, offering a compelling use case for RLHF in domain-specific text generation tasks such as software testing automation.

---

6. Memory Optimization

Memory optimization is a critical component of any model training pipeline, especially when working with large language models (LLMs) and real-time user feedback. In this implementation, memory optimization is achieved through various deliberate techniques including gradient clearing, cache flushing, and controlled tensor allocations. After each model update, torch.cuda.empty_cache() is invoked to clear unused memory buffers and prevent CUDA out-of-memory errors during the next inference or training step.

Additionally, inputs are truncated using the max_length parameter to ensure that tokenized prompts do not exceed GPU memory capacity. This prevents unnecessary computation and storage overhead, allowing more samples to be processed in each session. The input tensors are also

explicitly moved to the computing device (CPU or CUDA) to avoid memory mismatch issues, and unnecessary variables are discarded immediately after use to maintain optimal RAM and VRAM health.

Further memory control is provided by setting the PYTORCH_CUDA_ALLOC_CONF environment variable to expandable_segments:True. This experimental feature from PyTorch allows better memory reuse and reduces fragmentation during frequent allocation and deallocation operations. Combined with small batch sizes and frequent cache clearing, these optimizations significantly improve model responsiveness and stability, especially when deploying on limited hardware environments like Google Colab.

In essence, the code reflects a thoughtful strategy for balancing performance with resource limitations. These memory-aware practices are not only essential for avoiding runtime errors but also contribute to the feasibility of continuous interactive training in an RLHF pipeline. It enables users to run fine-tuning loops multiple times without needing high-end GPUs or restarting the runtime session frequently.

7. Model Saving and Reusability

Saving and reusing the fine-tuned model is an integral part of this implementation, ensuring that the improvements made through RLHF are persistent and can be deployed in future sessions. After completing the feedback loop, the model and

tokenizer are saved using the Hugging Face save_pretrained method into a specified directory (phi_finetuned1) on Google Drive. This allows seamless access and reuse in later stages without the need to re-train from scratch.

This model saving technique ensures version control and experiment tracking. Users can store multiple iterations of their fine-tuned models and compare them based on performance, feedback scores, and test case quality. Moreover, these saved models can be loaded onto different machines or shared with team members to validate and extend the use cases, promoting collaborative development in test automation.

Another benefit of saving the model is modular deployment. Once the model is trained with sufficient feedback, it can be integrated into a larger CI/CD pipeline for software testing, automating the generation of new test cases as requirements evolve. Additionally, inference time is significantly reduced when using a pre-saved fine-tuned model since it skips the training phase and directly loads the improved weights.

Overall, the reusability and portability of the trained model support sustainable AI practices. It makes the RLHF implementation practical for real-world applications, ensuring that user feedback doesn't go to waste and can continue contributing to long-term model performance enhancements in a scalable and efficient manner.

8. Benefits of RLHF in Software Testing

Integrating Reinforcement Learning with Human Feedback into software testing provides a range of unique advantages, particularly in automating the generation of comprehensive, context-aware, and high-precision test cases. The most significant benefit lies in the model's ability to evolve through direct feedback, meaning that it can learn and adapt to the testing standards, terminologies, and edge cases specific to the organization or project. This personalization leads to significantly improved relevance and usefulness of generated test cases.

Another advantage is the increased coverage and diversity of test cases. Through iterative feedback and fine-tuning, the model begins to understand and predict not just typical functional test cases but also edge cases, negative scenarios, and integration test paths that are often overlooked in manual or scripted generation techniques. This diversity leads to better risk identification and more robust software systems.

Additionally, RLHF allows for continuous learning and improvement without the need for manually labeled datasets. Unlike traditional supervised learning that requires large annotated corpora, RLHF simply uses scalar rewards from human reviewers to guide model training. This makes it feasible for smaller teams and organizations that lack the resources to curate large training datasets but still require intelligent test automation systems.

Finally, the human-in-the-loop design of RLHF introduces explainability and trust. By involving testers in the loop and showing them how their feedback is being utilized, the system builds user confidence and fosters a collaborative AI-testing environment. This dynamic feedback-training cycle aligns AI behavior closely with user expectations, creating a more transparent and effective automation solution for software quality assurance.

9. Test Case Generation Examples

Test case generation is a critical aspect of software testing, and when combined with RLHF, it becomes more iterative and refined. The process begins with the model generating test cases based on the input query, which could be related to functional testing, edge cases, performance, or other testing types. For example, given a query such as "Generate test cases for functional requirements," the model will first retrieve relevant chunks of information from the SRS document, followed by generating test cases aligned with these requirements.

Once generated, the test cases undergo human feedback. Feedback is provided on a scale of -1 (bad), 0 (neutral), and 1 (good), guiding the model to refine the test cases further. For instance, a test case that overlooks certain edge conditions or doesn't account for input variability may receive negative feedback, prompting the model to generate a more comprehensive test case in the next iteration. The feedback

loop allows the test cases to evolve iteratively, with each cycle improving their relevance and completeness.

This cycle of feedback and refinement continues until the generated test cases meet the expectations and specifications defined in the requirements. Over multiple iterations, the model's ability to generate contextually relevant, high-quality test cases increases, significantly reducing the need for manual intervention. These evolving test cases are more likely to capture the critical requirements and edge conditions for the system under test, improving test coverage and ensuring thorough validation of the software.

With RLHF, test case generation is dynamic and adaptive. As the model receives feedback, it continuously refines its understanding of what constitutes high-quality test cases. This iterative process helps in generating more targeted and precise test cases, reducing the chances of missing critical test scenarios and ensuring comprehensive test coverage for the software.

10. Visualization of Test Case Refinement

Visualization is a powerful tool to track how test cases evolve over time, especially in the context of RLHF. After each feedback cycle, the generated test cases are adjusted according to the provided feedback. Visualizing this refinement process helps testers and developers understand how each iteration improves the quality and relevance of the test cases. One effective visualization strategy could involve showing a flowchart or diagram of the feedback and refinement process,

illustrating the relationship between feedback, model updates, and the resulting changes in the test cases.

For example, a flowchart could display the stages of test case generation, where each feedback cycle refines the test cases further. The diagram would indicate how the model updates its outputs based on the feedback provided, showing the progression of test cases from initial drafts to more polished and contextually accurate scenarios. Additionally, a timeline could be used to demonstrate the iterative nature of RLHF, showing how test cases improve with each feedback round.

Moreover, it would be beneficial to visualize feedback scores (positive, neutral, negative) along with the corresponding changes in the test cases. Graphs and bar charts could illustrate the frequency of each feedback type and show how the model adapts over time. These visualizations would make it easier to assess the effectiveness of the feedback loop and the model’s learning process.

Such visualizations not only help in tracking the performance of the RLHF system but also aid in understanding how human feedback drives improvements in the generated test cases. This helps ensure that the test case generation process is transparent, and stakeholders can easily see the impact of their feedback on the model's outputs.

11. Evaluation Metrics for RLHF Performance

The success of RLHF-based test case generation can be measured using a variety of evaluation metrics. One of the most important metrics is **test case relevance**, which gauges

how well the generated test cases align with the software requirements specified in the SRS document. Relevance can be quantified by comparing the content of the test cases with the specific requirements outlined in the document, ensuring that each test case is addressing the key aspects of the software's functionality.

Coverage is another critical metric that evaluates whether the generated test cases address all possible scenarios, including edge cases and boundary conditions. Functional test cases should include a range of inputs, conditions, and workflows to ensure complete validation of the software. Similarly, performance test cases should assess the system’s response under varying load conditions, while security tests should cover potential vulnerabilities.

Additionally, **human feedback score** is a vital metric in the RLHF framework. It tracks the quality of feedback provided by human testers, which is essential for refining the generated test cases. By analyzing the distribution of feedback scores (positive, neutral, negative), developers can determine which aspects of test case generation need the most attention and improvement. This score also helps in evaluating the overall satisfaction with the generated test cases.

Lastly, **model performance improvement** is another key metric that assesses the extent to which the model's ability to generate high-quality test cases improves after each round of feedback. By comparing test case relevance, coverage, and human feedback scores before and after model updates, testers

can quantify how well the RLHF process is enhancing the test case generation capabilities of the model.

12. Limitations and Challenges

While RLHF-based test case generation offers significant benefits, there are also several limitations and challenges that need to be addressed. One of the primary challenges is the **dependence on high-quality feedback**. The effectiveness of RLHF is directly tied to the quality of the feedback provided by human testers. If feedback is inconsistent or unclear, it can negatively impact the model's learning process, leading to suboptimal test case generation.

Another challenge is **scalability**. As the number of feedback cycles increases, managing and processing the data can become computationally expensive. Storing and analyzing large volumes of feedback data requires substantial memory and processing power, which could lead to performance bottlenecks. Furthermore, the need for frequent model updates during each feedback cycle can place significant demands on computational resources, particularly when working with large models.

Generalization is also a concern. While RLHF allows the model to learn and improve over time, there is a risk that it may not generalize well across different software domains or projects. A model that works well for generating test cases for a particular type of software might not perform as effectively for others, especially if the software domains vary significantly. Fine-tuning the model for multiple software

domains requires large datasets and continuous adaptation, which can be resource-intensive.

Finally, there is the issue of **bias in feedback**. Since feedback is provided by humans, it may reflect individual biases, which could distort the model's learning process. For example, a tester might consistently favor certain test case types or overlook specific scenarios, leading to a skewed understanding of what constitutes a high-quality test case. Ensuring unbiased and diverse feedback is essential to maintain the integrity of the RLHF process.

13. Future Work and Improvements

While RLHF-based test case generation is promising, there are several avenues for future work and improvements that can make the system even more robust and effective. One area for enhancement is **increasing the model's ability to handle more complex requirements**. As software systems become more intricate, the ability of the model to understand and generate test cases for complex scenarios needs to be improved. This could involve incorporating more advanced models or combining RLHF with other techniques such as **few-shot learning** to handle diverse software requirements efficiently.

Another area of improvement is **scalability**. As the volume of software requirements grows, the system must be able to handle large-scale documents and produce test cases in a timely manner. This can be achieved through more efficient

data structures, parallel processing, or specialized hardware for faster processing. Additionally, there could be improvements in the **user feedback interface**. Providing a more intuitive and user-friendly method for human testers to give feedback could streamline the feedback loop and make the overall process more efficient.

Cross-domain adaptability is another challenge that requires attention. Currently, the system might be tailored to specific types of software or industries. Future work could focus on making the system adaptable to a wide range of software domains, from embedded systems to web applications. This would require the model to understand domain-specific terminologies and constraints, thereby improving its test case generation capabilities across different fields.

Lastly, **automated feedback collection** could be explored as an extension to the RLHF process. Instead of relying on manual feedback from human testers, the system could incorporate automated mechanisms for evaluating test case quality. This could involve using other machine learning techniques, such as supervised learning or unsupervised evaluation metrics, to provide an additional layer of feedback for further model refinement.

OUR RESULTS :

```
Feedback Round 1 of 10 - Generating Functional Test Cases:
Functional Test Cases:
- Test Case ID: TC-01
- Test Case Name: Verify user registration
- Test Steps:
  1. Navigate to the registration page on the OSS homepage.
  2. Enter a valid email and password in the respective fields.
  3. Click on the
Generated Test Case:
- Test Case ID: TC-01
- Test Case Name: Verify user registration
- Test Steps:
  1. Navigate to the registration page on the OSS homepage.
  2. Enter a valid email and password in the respective fields.
  3. Click on the
Enter feedback (1=good, 0=neutral, -1=bad): 0
Reward received: 0
Setting `pad_token_id` to `eos_token_id`:50256 for open-end generation.
Memory error during update: CUDA out of memory. Tried to allocate 50.00 MiB. GPU 0 has a total capacity of 14.74 GiB of which 30.12 MiB is free. Process 184002 has 14.71 GiB memory
```

```
Feedback Round 2 of 10 - Generating Edge Case Test Cases:
Edge Case Test Cases:
| Test Case ID | Test Case Description | Test Steps | Expected Result | Actual Result | Pass/Fail |
| --- | --- | --- | --- | --- | --- |
| TC-001 | User Registration |
  - Navigate to the OSS homepage and click
Generated Test Case:
| Test Case ID | Test Case Description | Test Steps | Expected Result | Actual Result | Pass/Fail |
| --- | --- | --- | --- | --- | --- |
| TC-001 | User Registration |
  - Navigate to the OSS homepage and click
Enter feedback (1=good, 0=neutral, -1=bad): 1
Setting `pad_token_id` to `eos_token_id`:50256 for open-end generation.
Reward received: 1
Memory error during update: CUDA out of memory. Tried to allocate 20.00 MiB. GPU 0 has a total capacity of 14.74 GiB of which 14.12 MiB is free. Process 184002 has 14.72 GiB memory i
```

```
Feedback Round 7 of 10 - Generating Regression Test Cases:
Regression Test Cases:
1. User Registration
- Test Case ID: TC-001
- Test Case Description: Verify that users can create an account with valid email and password.
- Precondition: The user is on the registration page.
- Test Steps:
  - Enter a valid email and password
Generated Test Case:
1. User Registration
- Test Case ID: TC-001
- Test Case Description: Verify that users can create an account with valid email and password.
- Precondition: The user is on the registration page.
- Test Steps:
  - Enter a valid email and password
Enter feedback (1=good, 0=neutral, -1=bad): 1
Setting `pad_token_id` to `eos_token_id`:50256 for open-end generation.
Reward received: 1
Memory error during update: CUDA out of memory. Tried to allocate 20.00 MiB. GPU 0 has a total capacity of 14.74 GiB of which 14.12 MiB is free. Process 184002 has 14.72 GiB memory

Feedback Round 8 of 10 - Generating Integration Test Cases:
Integration Test Cases:
1. Verify that users can create an account with a valid email and password.
2. Verify that users must verify their email address within 24 hours of creating an account.
3. Verify that passwords must be at least 8 characters long, including one uppercase letter and one number.
Generated Test Case:
1. Verify that users can create an account with a valid email and password.
2. Verify that users must verify their email address within 24 hours of creating an account.
3. Verify that passwords must be at least 8 characters long, including one uppercase letter and one number.
Enter feedback (1=good, 0=neutral, -1=bad): 1
Setting `pad_token_id` to `eos_token_id`:50256 for open-end generation.
Reward received: 1
Memory error during update: CUDA out of memory. Tried to allocate 20.00 MiB. GPU 0 has a total capacity of 14.74 GiB of which 14.12 MiB is free. Process 184002 has 14.72 GiB memory
```

```
Feedback Round 9 of 10 - Generating Negative Test Cases:
Negative Test Cases:
| Test Case ID | Test Case Description | Test Steps | Expected Result | Actual Result | Pass/Fail |
|------------|----------------------|-----------|----------------|--------------|----------|
| TC-01      | Verify user registration | 1. Navigate to
Generated Test Case:
| Test Case ID | Test Case Description | Test Steps | Expected Result | Actual Result | Pass/Fail |
|------------|----------------------|-----------|----------------|--------------|----------|
| TC-01      | Verify user registration | 1. Navigate to
Enter feedback (1=good, 0=neutral, -1=bad): 1
Setting `pad_token_id` to `eos_token_id`:50256 for open-end generation.
Reward received: 1
Memory error during update: CUDA out of memory. Tried to allocate 20.00 MiB. GPU 0 has a total capacity of 14.74 GiB of which 14.12 MiB is free. Process 184002 has 14.72 GiB memory

Feedback Round 10 of 10 - Generating Exploratory Test Cases:
Exploratory Test Cases:
| Test Case ID | Test Case Description | Test Steps | Expected Result | Actual Result | Status |
| ----------- | --------------------- | ---------- | --------------- | ------------- | ------ |
| TC1 | Verify user registration | Enter valid email and password, click on register button
Generated Test Case:
| Test Case ID | Test Case Description | Test Steps | Expected Result | Actual Result | Status |
| ----------- | --------------------- | ---------- | --------------- | ------------- | ------ |
| TC1 | Verify user registration | Enter valid email and password, click on register button
```

```
Completed 10 iterations. Model updated for 10 test case types.

Final Refined Test Cases (Last Type):
 | Test Case ID | Test Case Description | Test Steps | Expected Result | Actual Result | Status |
| ------------ | --------------------- | ---------- | --------------- | ------------- | ------ |
| TC1 | Verify user registration | Enter valid email and password, click on register button
Fine-tuned model saved successfully.
```

14. Conclusion

In conclusion, the integration of **Reinforcement Learning with Human Feedback (RLHF)** for test case generation represents a significant advancement in automating the software testing process. This approach allows for dynamic, iterative refinement of generated test cases, ensuring they are highly relevant and comprehensive in covering the various functional and non-functional requirements of the software under test. The continuous improvement through human feedback not only helps in addressing missing edge cases and scenarios but also enhances the quality of the overall testing process.

The feedback loop inherent in RLHF enables the model to adapt and improve over time, making it a powerful tool for generating high-quality test cases that meet real-world testing needs. This iterative refinement process helps reduce human intervention and manual labor, allowing testers to focus on more strategic tasks while ensuring that the generated test cases meet the required quality standards.

While there are challenges, including dependence on feedback quality, scalability, and generalization across software

domains, the benefits of using RLHF for test case generation far outweigh these hurdles. It brings a level of flexibility and customization that traditional automated test case generation techniques lack. The system becomes more intelligent with each feedback cycle, allowing it to handle increasingly complex software systems.

Looking ahead, there are exciting opportunities to expand and refine this approach. By addressing challenges such as cross-domain adaptability and scaling to larger datasets, the RLHF framework can become an essential tool in the software testing toolkit. The future of test case generation lies in leveraging the power of human feedback to continually refine and improve the quality of generated test cases, ultimately leading to more reliable and efficient software testing practices.