STACKFEED: Structured Textual Actor-Critic Knowledge base editing with FEEDback

Naman Gupta[* 1] **Shashank Kirtania**[* 2] **Priyanshu Gupta**[2]
Sumit Gulwani[3] **Arun Iyer**[1] **Suresh Parthasarathy**[1] **Arjun Radhakrishna**[3]
Sriram K. Rajamani[1] **Gustavo Soares**[3]
[1]Microsoft Research India, Bengaluru [2]Microsoft, Bengaluru [3]Microsoft, Redmond
{t-nagupta, t-skirtania, priyansgupta}@microsoft.com
{sumitg, ariy, supartha, arradha, sriram, gsoares}@microsoft.com

Abstract

Large Language Models (LLMs) often generate incorrect or outdated information, especially in low-resource settings or when dealing with private data. To address this, Retrieval-Augmented Generation (RAG) uses external knowledge bases (KBs), but these can also suffer from inaccuracies. We introduce STACKFEED, a novel **S**tructured **T**extual **A**ctor-**C**ritic **K**nowledge base editing with **FEED**back approach that iteratively refines the KB based on expert feedback using a multi-actor, centralized critic reinforcement learning framework. STACKFEED defines a ReACT actor agent on each document to perform structured edits based on document-specific targeted instructions. Experimental results showcase that STACKFEED significantly improves KB quality and performance of the RAG system. We evaluate STACKFEED on low-resource programming problems, modified python packaged and factual question-answering tasks.

1 Introduction

Large Language Models (LLMs) often produce incorrect or outdated information, particularly in low-resource settings or when handling private data. Even if the information provided is accurate, LLMs can generate hallucinated or imaginary content alongside it (Zhang et al., 2025; Maynez et al., 2020; Zhou et al., 2021). A promising solution to address these issues is the integration of retrieval components that extract relevant information from external knowledge sources, known as Retrieval-Augmented Generation (RAG) (Chen et al., 2017; Izacard et al., 2022; Shi et al., 2023; Li et al., 2025b). For clarity, we will refer to these external knowledge sources as Knowledge Bases (KBs). However, KBs themselves can suffer from inaccuracies, incompleteness, or outdated content. To address these challenges, there is growing interest in Knowledge Editing (KE) techniques to enhance LLMs with up-to-date and accurate knowledge.

Advancements in KE have focused on updating the model's parameters (De Cao et al., 2021a; Meng et al., 2022, 2023), adding new parameters to model (Huang et al., 2023; Yu et al., 2024), and holding additional memory (Madaan et al., 2022; Wang et al., 2024a,b). Contrary to approaches that update model parameters or add new parameters that require white-box access to LLMs, memory-based approaches can work with black-box access to LLMs. In a similar line of thought, recently, KE approaches have also focused on refining the KBs themselves (Li et al., 2025a). For example, the method proposed by Li et al. (2025a) continuously updates KBs with new information when presented with a document containing the exact information to be updated, such as updating the current identity of the British Prime Minister in the KB when the news of election results is provided. This approach demonstrates that directly editing the KB is more effective than simply adding new documents, which may coexist with outdated or inaccurate ones. Removing older documents is often not feasible, as only certain sections may be incorrect, while other parts could still provide valuable information for different queries.

However, in applications such as chatbots or code generation tools that rely on API documentation, up-to-date information may not always be readily available in structured documents (Ramjee et al., 2024; Afzal et al., 2024; Liu et al., 2025). In these scenarios, expert feedback becomes essential—not only for correcting erroneous outputs from the LLM but also for directly updating the underlying knowledge base (KB) with accurate information. This need is particularly pressing in live systems that depend on real-time, reliable data. Domains like healthcare (Ramjee et al., 2024), legal tech (Liu et al., 2025), and financial services demand high precision and instant updates. Ensuring continuous and trustworthy KB revisions is therefore critical to maintaining the safety, effectiveness, and reliability of RAG applications in such high-stakes environments.

To leverage expert or oracle feedback, we propose STACKFEED, a **S**tructured **T**extual **A**ctor-**C**ritic **K**nowledge base editing with **FEED**back technique. Our contributions are as follows:

1. **Introduction of Feedback-Driven KB Editing**: We present a novel framework that refines the knowledge base using structured edits based on oracle or expert feedback.
2. **Definition and Evaluation of KB Character-**

* Equal contribution.

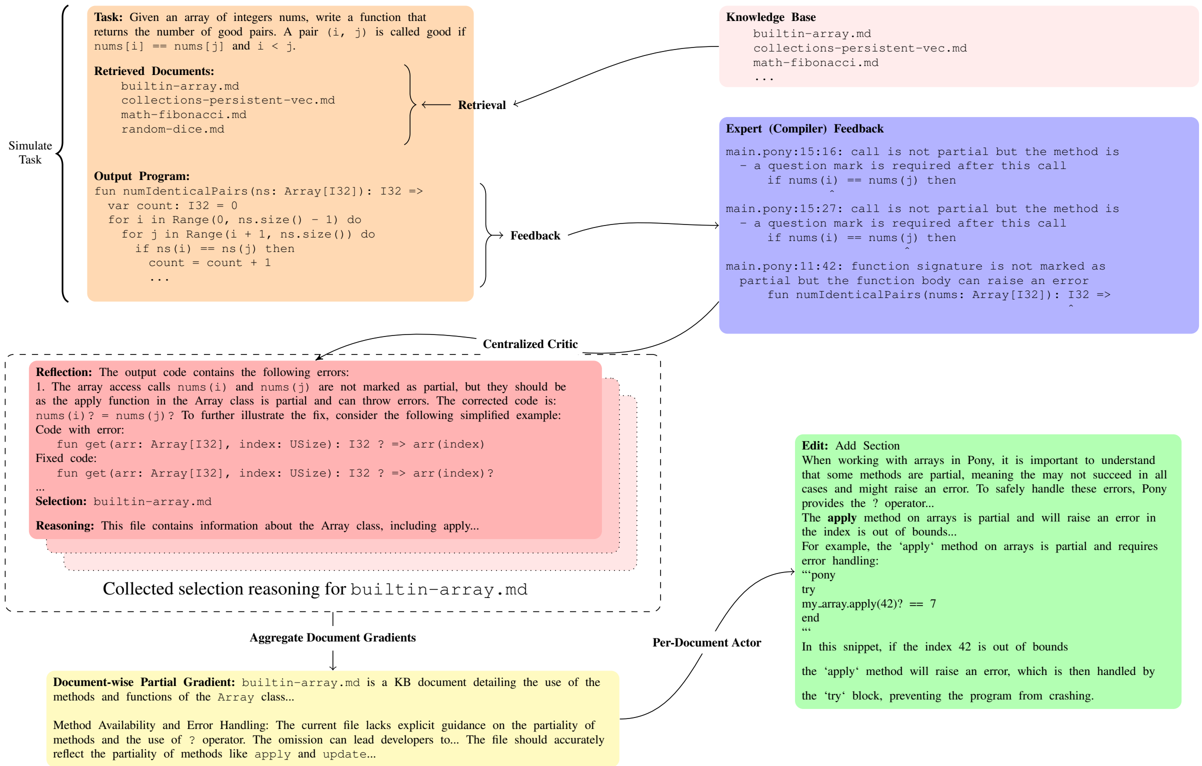

Figure 1: Example of the STACKFEED pipeline in the ARKS Pony scenario. We explain the example in more detail in appendix A.2

istics: We define desirable characteristics for knowledge base refinement, including coherence, completeness and introduce corresponding metrics to quantitatively assess these properties.

3. **Empirical Evaluation and Performance Gains**: We demonstrate that STACKFEED significantly improves the accuracy and reliability of RAG system in a variety of settings.

2 Related work

The STACKFEED framework addresses a key limitation of current RAG systems: the inability to dynamically update Knowledge Bases (KBs) without retraining or altering model parameters. Our work draws from research in Retrieval-Augmented Generation (RAG), Continual Learning and incorporating insights from Multi-Agent Reinforcement Learning (MARL) to propose an effective solution for KB editing.

Retrieval Augmented Generation (RAG): RAG systems enhance LMs by retrieving relevant knowledge from a KB based on the input query and appending it to the context, thereby addressing the limitations of standalone LMs that lack sufficient context and produce inaccurate answers (Chen et al., 2017; Khandelwal et al., 2020; Guu et al., 2020; Izacard et al., 2022; Shi et al., 2023). These systems dynamically construct contexts from unstructured KBs without modifying the LM's internal parameters. STACKFEED further enhances RAG systems by refining the KB itself based on feedback, ensuring more accurate and up-to-date information. Recent works showcase the failure of RAG due to inconsistencies in the retrieved documents (Xiang et al., 2024; Wang et al., 2025).

Knowledge Editing: Knowledge Editing approaches fall into two categories: **Model Editing**, which modifies the LM parameters directly, and **Input Editing**, which updates the knowledge supplied to the model. While Model Editing efficiently alters specific facts using specialized secondary models or altering parameters (De Cao et al., 2021b; Meng et al., 2023), it struggles to ensure consistent updates across contexts (Onoe et al., 2023; Hua et al., 2024). In contrast, Input Editing modifies the KB itself, enabling updates to be reflected in outputs without changing model parameters (Wang et al., 2024b; Li et al., 2025a). STACKFEED builds on input editing techniques by leveraging expert feedback to refine the KB systematically, ensuring more accurate and consistent responses.

Prompt Optimization: With the advent of LMs, some recent works approximate gradients in text-based environments using LMs (Pryzant et al., 2023; Wang et al., 2023; Kirtania et al., 2024; Gupta et al., 2024) for optimizing task prompts. STACKFEED is inspired by these approaches and generates textual reflections, similar to MetaReflection (Gupta et al., 2024) and (Shinn et al., 2023), as proxies for gradients. It provides actionable guidance for document updates without the need for differentiable models. Additionally, STACKFEED adopts clustering strategies for feedback aggregation from works like UniPrompt (Juneja et al., 2024)- ensuring that actors receive coherent and non-redundant instructions.

3 Problem Formulation

Typical RAG systems assume that the information present in the retrieved documents is correct and consistent. Our work focuses on scenarios where incorrect answers are generated due to issues in the retrieved documents from a Knowledge Base ($\mathcal{K}$).

More formally, we define the $\mathcal{K}$ as a collection of documents D_i for $i = 1, \ldots, N$. Each document D_i can be represented as a set of chunks c_{ij}. The state of $\mathcal{K}$ is the specific configuration of all chunks within it.

For a given query q, such as the code generation Task in Figure 1, a retriever fetches a set of relevant documents $\Gamma(q, \mathcal{K})$. In the example, this corresponds to the Retrieved Documents list, which includes `builtin-array.md` and `collections-persistent-vec.md`.

An LLM, M, then generates a response r based on the query and the retrieved documents, i.e., $r = M(q, \Gamma(q, \mathcal{K}))$. The initial Output Program in Figure 1 is an instance of such a response.

However, this response r may be incorrect. We obtain feedback on the correctness of r, for instance, from Expert (Compiler) Feedback as shown in Figure 1. This feedback reveals flaws in the generated program that stem from deficiencies in $\mathcal{K}$. The expert is used as a scoring function, g which evaluates whether a response r is correct or not.

Our goal is to optimize the Knowledge Base state $\mathcal{K}$ to state $\mathcal{K}*$ which maximizes the scores for the responses for a batch of queries $\mathcal{Q}$ by learning from expert feedback:

$$\mathcal{K}^* = \arg\max_{\mathcal{K}} \frac{1}{|\mathcal{Q}|} \sum_{q_i \in \mathcal{Q}} g(\mathcal{M}(q_i, \Gamma(q_i, \mathcal{K}))) \quad (1)$$

4 Methodology

We propose STACKFEED, an agent that employs Monte Carlo Tree Search (MCTS) to search for an optimal state of the knowledge base (KB). STACKFEED utilizes a *multi-actor*, *centralized critic* reinforcement learning architecture to guide transitions between KB states. This setup enables efficient exploration of a large, structured edit space (Wang et al., 2023; Gupta et al., 2024), facilitating strategic and interpretable knowledge refinement.

The use of a multi-actor architecture with a centralized critic is central to STACKFEED's design. In this formulation, each document within the KB is managed by a dedicated actor responsible for localized edits, while a centralized critic provides joint feedback by aggregating signals across all actor-document interactions. Our design aligns with recent findings from Lyu et al. (2024), who show that *history–state centralized critics*—critics conditioned on both joint observation histories and global state—can offer accurate and stable policy gradients, particularly in settings with partial observability and distributed decision-making.

In our context, where agents must coordinate to revise distinct portions of a shared KB based on sparse oracle feedback, this centralized view enables effective credit assignment. Specifically, we incorporate mechanisms inspired by counterfactual multi-agent policy gradients (COMA) (Foerster et al., 2018) to attribute responsibility for errors to specific documents and actors. The centralized critic leverages these credit signals to generate high-quality textual reflections that guide each actor's editing trajectory. This design allows STACKFEED to maintain coherence across KB documents while iteratively improving task-specific correctness and completeness.

By combining decentralized editing with centralized, feedback-driven evaluation, STACKFEED learns to make targeted and interpretable updates to the KB. This architecture is particularly suited for real-world retrieval-augmented generation (RAG) settings, where maintaining consistency and relevance across independently edited documents is critical for downstream performance.

4.1 Knowledge base editing as state search

We model knowledge base (KB) editing as a state optimization problem over the configuration of documents in the KB. Given a query and retrieved evidence, a language model generates a response. When errors arise due to incomplete or inaccurate evidence, we update the KB such that future responses are more accurate.

In this formulation, each KB state reflects a specific configuration of document contents, and actions correspond to edits applied to these documents. A transition function updates the KB by applying these edits, and a reward function evaluates the quality of the resulting KB state based on how well it supports correct and complete responses across a set of queries.

Our objective is to find the optimal KB state that maximizes this reward. This enables a feedback-driven editing process where the system learns to apply targeted modifications to improve RAG performance in a data-driven and interpretable manner. We define the action space, search space and the optimization objective more formally in the appendix section A.1.

4.2 Knowledge Base Editing Agent

We define KB editing agent that operates on a reward signal as a model's performance over a batch of queries for the given knowledge base in a RAG system.

Centralized Critic: The centralized critic C evaluates the RAG system's performance by analyzing expert feedback and the current knowledge base state. When errors occur, the critic identifies which specific documents caused the problems and generates targeted feedback for improvement.

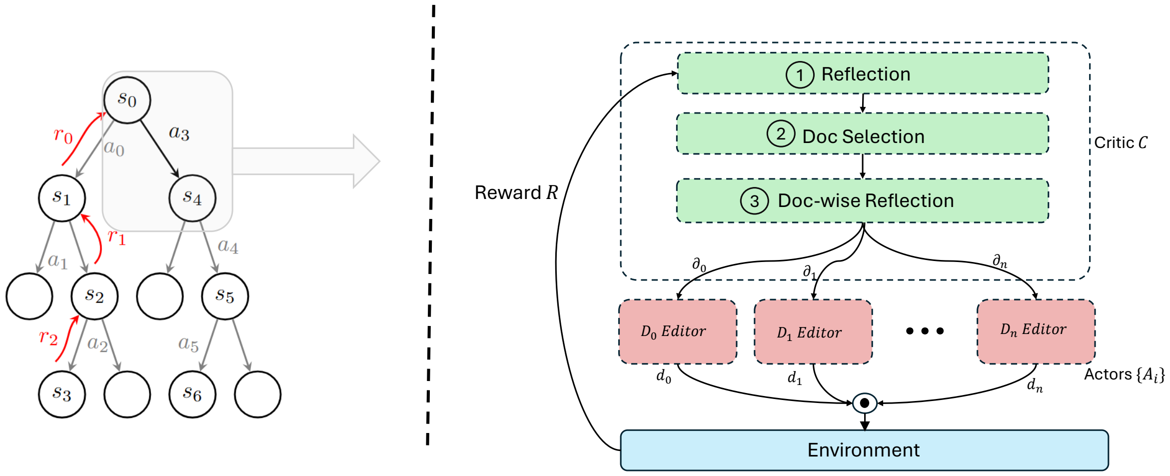

Figure 2: a) MCTS (Monte Carlo Tree Search) planning for state search. The tree structure enables strategic planning for STACKFEED. b) A simplified state transition example. Upon receiving a reward from the environment (or expert) on the given state of the knowledge base (KB) s_0, a centralized critic 1 generates a reflection on observed failures to calculate the textual gradient. The critic uses this reflection to select documents responsible for the error and 2 assigns credit to actors in the form of document-wise reflections. The actors then iteratively edit the documents to reach state s_4.

The critic examines failed queries to determine which retrieved documents led to incorrect answers. Following established methods from prior work (Pryzant et al., 2023; Juneja et al., 2024; Gupta et al., 2024), it uses LLMs to create text-based gradients highlighting the errors. Rather than simply listing all issues, the critic clusters similar problems across multiple queries to identify common patterns and generate more generalizable insights.

These aggregated reflections are analogs to partial gradients ∂ for each document that guide each document-specific actor A_j on improving their assigned documents. This approach ensures document updates address systematic issues rather than isolated errors, leading to more effective knowledge base refinement. By analyzing failures across multiple queries and clustering similar issues, the critic provides more strategic guidance than treating each error in isolation.

Actors: Each document $D_i \in \mathcal{K}$ is managed by a distinct actor, A_i, which is modeled as a ReACT agent (Yao et al., 2023) responsible for making structured edits to its document. Each actor operates independently, receiving reflections from the centralized critic on how to modify the content of $D_i = [c_{ij}]$. The actors need to only update these chunks as needed. Each actor is provided with a set of parametrized actions to perform precise edits to the document chunks, allowing for flexible and context-specific edits. The set of possible actions includes:

- **Edit Chunk**: Modifies an existing chunk within a document by replacing content with updated text.
- **Add Chunk**: Creates a new chunk with specified content and adds it to the document.
- **Delete Chunk**: Removes an existing chunk from the document entirely.

The ReACT agent utilizes these reflections and iteratively generates a trajectory $t_0 = a_0, a_1, a_2 \cdot a_n$ of edit actions to the document until the errors are resolved or the knowledge gaps are filled. This controlled editing process improves the accuracy of the RAG system by ensuring that the KB contains up-to-date and relevant information. After the completion of the actor runs, we generate the edit diffs for each document d_i and pool them to generate the KB edit action $u = [d_i]_{i=1}^{|\mathcal{K}|}$

5 Experimental Setup

5.1 Baseline

While there has been a rich body of works in the area of prompt optimization, to the best of our knowledge, STACKFEED is the first work targeting the feedback-driven textual Knowledge Base Editing problem. Therefore, to perform a holistic evaluation of STACKFEED we implement - PROMPTAGENT-E, an extension of PROMPTAGENT (Wang et al., 2023) for the KB editing task. PROMPTAGENT formulates prompt optimization as a strategic planning problem using Monte Carlo Tree Search (MCTS). We have described our implementation on top of PROMPTAGENT in appendix section A.3

5.2 Datasets

Knowledge Base Editing can be useful for scenarios where the KB is either incomplete or incorrect. We evaluate on EVOR (Su et al., 2024) which is a dataset of documentation for programming language *Pony* which

Dataset	Pony		SciPy		Tensorflow		CLARK-news	
	Acc	σ	Acc	σ	Acc	σ	Acc	σ
Base KB GPT-4Turbo	29.99	1.57	52.04	0.00	28.88	2.18	26.27	1.20
Base KB GPT-4o	31.41	1.28	54.13	1.22	31.75	2.91	28.80	1.69
Base KB GPT-4.1	35.40	2.52	53.40	2.43	34.60	3.11	30.89	1.20
PROMPTAGENT-E	32.22	1.57	53.40	3.12	47.77	3.57	28.80	2.39
STACKFEED GPT-4Turbo	37.04	1.28	59.38	1.22	53.84	3.11	37.28	1.69
STACKFEED GPT-4o	42.32	2.11	**61.60**	2.43	55.32	2.18	40.40	1.63
STACKFEED GPT-4.1	**45.62**	3.67	60.83	2.84	**57.61**	2.18	**43.03**	2.14

Table 1: Correctness performance comparison between STACKFEED and baseline models across multiple datasets, reported as accuracy percentages (higher is better).

can be incomplete in details along with natural language to code questions in them. Similarly, it has two more datasets about custom versions of *SciPy* and *Tensorflow* with the original documentation of these libraries which must be adapted for these custom versions. We also use the also CLARK-News dataset (Li et al., 2025a) which is a natural language dataset of news articles of outdated factual information. We describe each dataset in detail in the appendix in A.4.

5.3 System Configurations

For our experiments, we set a maximum search depth of 3, an expansion width of 3, and a maximum of 5 iterations. The UCT algorithm with an exploration constant of 2.5 is used for expansion nodes. The parameters are chosen to balance between effective exploration and computational cost.

We set up a generic RAG system that uses an embedding similarity for semantic retrieval. Additionally, in line with prior works like (Zhang et al., 2023) for coding-related tasks, we use an iterative retrieval setup wherein we first generate a code using naive retrieval and then query the database again with both the question and the generated code to improve the quality of retrieval before generating the final result. We use OPENAI-TEXT-EMBEDDING-3-LARGE as the embedding model and use cosine similarity as a metric of embedding match for ranking.

5.4 Metrics

Correctness We evaluate the correctness of the KB by evaluating it on a *test set* queries on respective tasks. This separation of the test-train set of queries reduces the risk of contamination of the examples and falsified improvements in performance. We also define two metrics *completeness* and *coherence* to understand the quality of the edits made by STACKFEED.

Completeness A knowledge base should be *complete* with respect to the task, that means it should contain all the information necessary to assist RAG system for task at hand. Given the open-ended nature of tasks that typical RAG agents are designed for, it is hard to quantify a closed-form metric of *completeness*. However, an ideal KB editing system should at least be able to incorporate external feedback well. To evaluate this we use the precision *train set* to estimate the degree of expert feedback incorporated in the learned KB.

Coherence Given the semantic and textual nature of the Knowledge Base, it is important that the documents in the Knowledge base are coherent and consistent even after editing. This not only makes the document interpretable for human consumption, it also help reduce in-context noise during LLM inference, which has been shown to affect LLM performance (Liu et al., 2024). To quantify the degree of coherence of the KB, we first calculate coherence scores for each edited document using G-Eval (Liu et al., 2023). We use the G-eval score to gauge the coherence of an edit made to a KB document to the document itself. And the mean of this document-level coherence over all the documents is defined as the coherence of an edited KB.

6 Results and Analysis

We evaluate STACKFEED on three different OpenAI models *GPT-4Turbo*, *GPT-4o* and *GPT-4.1*[1] on three different configurations. Firstly, evaluating the performance on the Base KB, this is the initial state of the KB s_0 without any edits. We then make a series of edits on s_0 using PROMPTAGENT-E and a series of edits by STACKFEED on KB state s_0.

We report our main results in table 1. We observe the performance of the RAG system constantly improve with edits done by STACKFEED.

Quality of edits As seen in Table 2, STACKFEED produces edits with a coherence score of 4 or higher. For KBs that need long-term maintenance (such as language and code documentation as seen in the Evor datasets), STACKFEED makes more coherent edits compared to the baseline. This is especially true for long documents, as seen in the EVOR Pony dataset. We also note that the edits

[1]https://platform.openai.com/docs/models

	Completeness (in %)				Coherence (1→5, higher is better)			
Dataset	Pony	SciPy	Tensorflow	CLARK-news	Pony	SciPy	Tensorflow	CLARK-news
PROMPTAGENT-E	3.22	33.33	33.33	11.86	1.86	2.0	4.0	1
STACKFEED GPT-4Turbo	9.68	31.38	44.44	13.79	4.6	4.30	**4.0**	1
STACKFEED GPT-4o	11.38	36.67	50.12	15.41	**4.67**	**4.67**	**4.0**	**2.33**
STACKFEED GPT-4.1	**13.45**	**41.46**	**52.24**	**18.62**	4.6	4.00	3.67	1

Table 2: Completeness and coherence comparison between STACKFEED and baseline models across multiple datasets. Completeness is reported as accuracy percentages (higher is better), while coherence is measured on a scale of 1-5 (higher is better).

	Single Edit	Greedy Search	MCTS
Correctness	36.14	41.34	45.62
Completeness	9.68	13.96	13.45
Coherence	4	3.34	4.6

Table 3: Comparison between single edit, greedy search and MCTS, on EVOR Pony dataset with GPT-4.1. We used max width=3 and max depth=5.

made by PROMPTAGENT-E were made in incorrect documents leading to more noisy generations.
We also observe that PROMPTAGENT-E added irrelevant section in example A.5 on the *lineSearch* and *norm_ppf* functions in a document about sparse matrices. These edits were made because the document was retrieved for questions which had errors regarding these functions. These edits are irrelevant to the document and showcase reason for failure in PROMPTAGENT-E is to make documents less coherent.

On the other hand, STACKFEED makes more relevant edits to the document which are contained to the context of the document. This shows how STACKFEED is able to maintain the coherence of the document in its edits. We demonstrate an complete example of edits made in appendix A.5
For a news-article-like dataset like CLARK-news with factual edits. Incoherency is naturally induced when the facts of the article change. In this dataset, coherence is sacrificed in bringing the facts of the article up to date, which is required to improve accuracy.

We also evaluate performance on EVOR-Pony dataset using just a single STACKFEED edit and greedy search in table 3. In greedy search, we greedily pick the most rewarding node at a particular depth. We observe even though the completeness of the document is quite similar the edits are much less coherent and the generalize less than MCTS based search.

7 Conclusion

We introduced STACKFEED, a novel framework for refining Knowledge Bases (KBs) in Retrieval-Augmented Generation (RAG) systems using a multi-actor, centralized critic architecture. STACKFEED enables efficient KB updates without retraining or altering model parameters by leveraging feedback-driven structured edits and textual gradients.

Our approach achieved superior performance in preserving knowledge base in terms of coherence, consistency, and completeness, resulting in enhanced performance of RAG system.

Broader Impact

The deployment of Retrieval-Augmented Generation (RAG) systems in real-world applications such as AI-powered developer assistants, enterprise chatbots, and domain-specific information retrieval relies heavily on the correctness and reliability of the underlying knowledge bases (KBs). However, maintaining these KBs is a persistent bottleneck due to frequent changes in domain-specific knowledge and the lack of automated mechanisms for continuous KB refinement. Our proposed system, **STACKFEED**, addresses this challenge through a feedback-driven framework for automatic knowledge base editing that learns from expert or oracle feedback in real-time deployments.

Our design introduces a modular, actor-critic architecture that can be integrated into existing pipelines with minimal engineering overhead. By defining clear KB quality metrics—*correctness*, *coherence*, and *completeness*. Our system provides actionable insights for both developers and auditors. This supports not only continuous improvement of deployed systems, but also regulatory compliance and human-in-the-loop oversight in high-stakes domains like healthcare, finance, and legal automation. By automating feedback incorporation into KBs, we reduce human maintenance cost, lower response errors in production RAG systems, and promote safer, more trustworthy deployments.

In industrial RAG based applications, post-deployment error-correction and maintenance is done through improving the quality of the retrieval system. We introduce another axis by enabling the optimization of the Knowledge base itself. It also paves the way for joint optimization of both the knowledge base and the retrieval mechanism, offering a more holistic and scalable solution to long-term system maintenance.

We hope this work paves the way for future industry adoption of learning-enabled infrastructure that maintains and improves itself in deployment, and encourages further exploration of editable memory systems as an alternative to end-to-end retraining for knowledge maintenance.

References

Anum Afzal, Alexander Kowsik, Rajna Fani, and Florian Matthes. 2024. Towards optimizing and evaluating a retrieval augmented qa chatbot using llms with human in the loop.

Danqi Chen, Adam Fisch, Jason Weston, and Antoine Bordes. 2017. Reading Wikipedia to answer open-domain questions. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1870–1879, Vancouver, Canada. Association for Computational Linguistics.

Nicola De Cao, Wilker Aziz, and Ivan Titov. 2021a. Editing factual knowledge in language models. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 6491–6506, Online and Punta Cana, Dominican Republic. Association for Computational Linguistics.

Nicola De Cao, Wilker Aziz, and Ivan Titov. 2021b. Editing factual knowledge in language models. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 6491–6506, Online and Punta Cana, Dominican Republic. Association for Computational Linguistics.

Jakob Foerster, Gregory Farquhar, Triantafyllos Afouras, Nantas Nardelli, and Shimon Whiteson. 2018. Counterfactual multi-agent policy gradients. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32.

Jakob N. Foerster, Gregory Farquhar, Triantafyllos Afouras, Nantas Nardelli, and Shimon Whiteson. 2017. Counterfactual multi-agent policy gradients. In *AAAI Conference on Artificial Intelligence*.

Priyanshu Gupta, Shashank Kirtania, Ananya Singha, Sumit Gulwani, Arjun Radhakrishna, Sherry Shi, and Gustavo Soares. 2024. Metareflection: Learning instructions for language agents using past reflections.

Kelvin Guu, Kenton Lee, Zora Tung, Panupong Pasupat, and Ming-Wei Chang. 2020. Realm: retrieval-augmented language model pre-training. In *Proceedings of the 37th International Conference on Machine Learning*, ICML'20. JMLR.org.

Wenyue Hua, Jiang Guo, Mingwen Dong, Henghui Zhu, Patrick Ng, and Zhiguo Wang. 2024. Propagation and pitfalls: Reasoning-based assessment of knowledge editing through counterfactual tasks. In *Findings of the Association for Computational Linguistics ACL 2024*, pages 12503–12525, Bangkok, Thailand and virtual meeting. Association for Computational Linguistics.

Zeyu Huang, Yikang Shen, Xiaofeng Zhang, Jie Zhou, Wenge Rong, and Zhang Xiong. 2023. Transformer-patcher: One mistake worth one neuron. In *The Eleventh International Conference on Learning Representations*.

Gautier Izacard, Patrick Lewis, Maria Lomeli, Lucas Hosseini, Fabio Petroni, Timo Schick, Jane Dwivedi-Yu, Armand Joulin, Sebastian Riedel, and Edouard Grave. 2022. Atlas: Few-shot learning with retrieval augmented language models.

Gurusha Juneja, Nagarajan Natarajan, Hua Li, Jian Jiao, and Amit Sharma. 2024. Task facet learning: A structured approach to prompt optimization.

Urvashi Khandelwal, Omer Levy, Dan Jurafsky, Luke Zettlemoyer, and Mike Lewis. 2020. Generalization through memorization: Nearest neighbor language models. In *International Conference on Learning Representations*.

Shashank Kirtania, Priyanshu Gupta, and Arjun Radhakrishna. 2024. LOGIC-LM++: Multi-step refinement for symbolic formulations. In *Proceedings of the 2nd Workshop on Natural Language Reasoning and Structured Explanations (@ACL 2024)*, pages 56–63, Bangkok, Thailand. Association for Computational Linguistics.

Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Scott Wen tau Yih, Daniel Fried, Sida Wang, and Tao Yu. 2022. Ds-1000: A natural and reliable benchmark for data science code generation. *ArXiv*, abs/2211.11501.

Belinda Z. Li, Emmy Liu, Alexis Ross, Abbas Zeitoun, Graham Neubig, and Jacob Andreas. 2025a. Language modeling with editable external knowledge. In *Findings of the Association for Computational Linguistics: NAACL 2025*, pages 3070–3090, Albuquerque, New Mexico. Association for Computational Linguistics.

Xiaopeng Li, Pengyue Jia, Derong Xu, Yi Wen, Yingyi Zhang, Wenlin Zhang, Wanyu Wang, Yichao Wang, Zhaocheng Du, Xiangyang Li, Yong Liu, Huifeng Guo, Ruiming Tang, and Xiangyu Zhao. 2025b. A survey of personalization: From rag to agent.

Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2024. Lost in the middle: How language models use long contexts. *Transactions of the Association for Computational Linguistics*, 12:157–173.

Yang Liu, Dan Iter, Yichong Xu, Shuo Wang, Ruochen Xu, and Chenguang Zhu. 2023. G-eval: Nlg evaluation using gpt-4 with better human alignment. In *Conference on Empirical Methods in Natural Language Processing*.

Zeyu Leo Liu, Shrey Pandit, Xi Ye, Eunsol Choi, and Greg Durrett. 2025. Codeupdatearena: Benchmarking knowledge editing on api updates.

Xiaotian Lyu, Alessandro Baisero, Yiqin Xiao, Brendan Daley, and Christopher Amato. 2024. On centralized critics in multi-agent reinforcement learning. *arXiv preprint arXiv:2408.14597*.

Aman Madaan, Niket Tandon, Peter Clark, and Yiming Yang. 2022. Memory-assisted prompt editing to improve GPT-3 after deployment. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 2833–2861, Abu Dhabi, United Arab Emirates. Association for Computational Linguistics.

Joshua Maynez, Shashi Narayan, Bernd Bohnet, and Ryan McDonald. 2020. On faithfulness and factuality in abstractive summarization. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 1906–1919, Online. Association for Computational Linguistics.

Kevin Meng, David Bau, Alex Andonian, and Yonatan Belinkov. 2022. Locating and editing factual associations in GPT. *Advances in Neural Information Processing Systems*, 36. ArXiv:2202.05262.

Kevin Meng, Arnab Sen Sharma, Alex J Andonian, Yonatan Belinkov, and David Bau. 2023. Mass-editing memory in a transformer. In *The Eleventh International Conference on Learning Representations*.

Yasumasa Onoe, Michael Zhang, Shankar Padmanabhan, Greg Durrett, and Eunsol Choi. 2023. Can LMs learn new entities from descriptions? challenges in propagating injected knowledge. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 5469–5485, Toronto, Canada. Association for Computational Linguistics.

Reid Pryzant, Dan Iter, Jerry Li, Yin Lee, Chenguang Zhu, and Michael Zeng. 2023. Automatic prompt optimization with “gradient descent” and beam search. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 7957–7968, Singapore. Association for Computational Linguistics.

Pragnya Ramjee, Bhuvan Sachdeva, Satvik Golechha, Shreyas Kulkarni, Geeta Fulari, Kaushik Murali, and Mohit Jain. 2024. Cataractbot: An llm-powered expert-in-the-loop chatbot for cataract patients.

Weijia Shi, Sewon Min, Michihiro Yasunaga, Minjoon Seo, Rich James, Mike Lewis, Luke Zettlemoyer, and Wen tau Yih. 2023. Replug: Retrieval-augmented black-box language models.

Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning.

Hongjin Su, Shuyang Jiang, Yuhang Lai, Haoyuan Wu, Boao Shi, Che Liu, Qian Liu, and Tao Yu. 2024. Arks: Active retrieval in knowledge soup for code generation. *ArXiv*, abs/2402.12317.

Fei Wang, Xingchen Wan, Ruoxi Sun, Jiefeng Chen, and Sercan Ö. Arık. 2025. Astute rag: Overcoming imperfect retrieval augmentation and knowledge conflicts for large language models.

Jiaan Wang, Yunlong Liang, Zengkui Sun, Yuxuan Cao, Jiarong Xu, and Fandong Meng. 2024a. Cross-lingual knowledge editing in large language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 11676–11686, Bangkok, Thailand. Association for Computational Linguistics.

Weixuan Wang, Barry Haddow, and Alexandra Birch. 2024b. Retrieval-augmented multilingual knowledge editing. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 335–354, Bangkok, Thailand. Association for Computational Linguistics.

Xinyuan Wang, Chenxi Li, Zhen Wang, Fan Bai, Haotian Luo, Jiayou Zhang, Nebojsa Jojic, Eric P. Xing, and Zhiting Hu. 2023. Promptagent: Strategic planning with language models enables expert-level prompt optimization.

Chong Xiang, Tong Wu, Zexuan Zhong, David Wagner, Danqi Chen, and Prateek Mittal. 2024. Certifiably robust rag against retrieval corruption.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. React: Synergizing reasoning and acting in language models.

Lang Yu, Qin Chen, Jie Zhou, and Liang He. 2024. Melo: Enhancing model editing with neuron-indexed dynamic lora. *Proceedings of the AAAI Conference on Artificial Intelligence*, 38(17):19449–19457.

Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. 2023. Repocoder: Repository-level code completion through iterative retrieval and generation.

Ziyao Zhang, Chong Wang, Yanlin Wang, Ensheng Shi, Yuchi Ma, Wanjun Zhong, Jiachi Chen, Mingzhi Mao, and Zibin Zheng. 2025. Llm hallucinations in practical code generation: Phenomena, mechanism, and mitigation. *Proc. ACM Softw. Eng.*, 2(ISSTA).

Chunting Zhou, Graham Neubig, Jiatao Gu, Mona Diab, Francisco Guzmán, Luke Zettlemoyer, and Marjan Ghazvininejad. 2021. Detecting hallucinated content in conditional neural sequence generation. In *Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021*, pages 1393–1404, Online. Association for Computational Linguistics.

A Appendix

A.1 Knowledge Base Editing as State Search

In our problem setting, the Knowledge Base ($\mathcal{K}$) is defined as a collection of documents $\mathcal{K} = \{D_i\}_{i=1}^n$. We assume each document consists of a number of chunks of text and can be represented as $D_i = [c_{ij}]$. The state $s \in \mathcal{S}$ of the system is represented by the current configuration of the KB, i.e., the content of all documents in $\mathcal{K}$.

Given a query q_i and a set of retrieved documents $\Gamma(q_i, \mathcal{K})$, the LLM $\mathcal{M}$ generates an answer o_i. When errors arise due to incomplete or incorrect information in the retrieved documents, our goal is to identify the optimal configuration of $\mathcal{K}$ that improves the accuracy of the system's responses. Thus, we define our state search problem as finding the best state s^* of the KB.

State Space: The state space $\mathcal{S}$ encompasses all possible configurations of the KB. Each state s corresponds to a particular set of document contents, represented as: $s = \{D_i\}_{i=1}^n$, where D_i denotes the content of document i and n is the number of documents in $\mathcal{K}$. The state s captures the overall structure and content of the KB at any given point. We set $s_0 = \mathcal{K}$.

State Transition Function: The state transition function $\mathcal{T}(s, u)$ defines how the KB changes in response to the action u taken by the agent. Each action contains modifications to one or more documents within the KB, resulting in a new KB configuration. The state transition is formalized as: $s' = \mathcal{T}(s, u)$, where s' is the new state of the KB after applying u.

Action Space: The action space $\mathcal{A}$ consists of list of diffs d_i corresponding to each document D_i. Essentially, $u = [d_i]_{i=1}^{|\mathcal{K}|}$.

Environment: We model the environment simply as a "patch" function, that takes the diff generated by the agent and patches the KB to produce the new state.

Optimization Objective: Following Equation 1, our objective then is to find the optimal state s^* of the KB that maximizes the overall performance of the RAG system, as measured by a global reward function R. The optimization problem is formulated as:

$$R(s) = \frac{1}{|Q|} \sum_{q_i \in Q} g(\mathcal{M}(q_i, \Gamma(q_i, s))) \tag{2}$$

$$s^* = \arg\max_{s \in \mathcal{S}} R(s) \tag{3}$$

where $R(s)$ represents the cumulative reward of the KB state s, reflecting its ability to support accurate and complete responses for a set of queries.

The reward function $R(s)$ is derived from the expert feedback on the system's generated answers and captures improvements in the KB. By optimizing for s^*, we ensure that the final state of the KB maximizes the overall accuracy and effectiveness of the RAG system, rather than focusing on an intermediate sequence of state transitions.

In summary, the state search formulation defines the problem of finding the optimal state s^* of the KB that maximizes the system's performance. This approach enables us to make targeted, feedback-driven edits to the KB and achieve a refined, high-quality knowledge base that better supports accurate answer generation.

A.2 Example and Overview

Figure 2 illustrates our technique applied to the Pony (Su et al., 2024), where a knowledge base (KB) for the low-resource programming language Pony supports a natural language-to-code task. Due to Pony's rarity, language models often generate code that fails to compile. To address this, we use the Pony compiler as an expert to provide feedback in the form of compile errors.

① *Evaluating the Knowledge Base State*: We start with an initial KB, including documents like `builtin-array.md`. The system retrieves relevant documents based on the given task (e.g., counting non-inversions in an array) and generates a program, which is evaluated by the compiler, resulting in feedback (e.g., compile errors).

② *Centralized Critic Analysis*: For all errors, the critic analyzes why the error occurred. For instance, if the `apply` method in the `Array` class is partial and may raise an error, the critic suggests adding a `?` to handle potential failures. Based on this reflection, the critic identifies which of the retrieved document is relevant for the error and provides a tailored reasoning for it.

③ *Per-Document Actor*: For each document in the KB, the gradients associated to it are aggregated. This aggregate gradient is used as a signal by the Per-Document Actor, in this case, the actor for document `builtin-array.md` to make edits to the document.

④ *Re-evaluation and MCTS Search*: After edits are applied, the KB is re-evaluated, generating new feedback and a reward score. This score guides a Monte Carlo Tree Search (MCTS) to explore different states of the KB, iterating through steps ①-③ to progressively refine the KB and improve the system's overall performance.

A.3 PromptAgent-E Baseline

PROMPTAGENT (Wang et al., 2023) is a technique developed for optimizing a single prompt. **PromptAgent-E** extends this approach to knowledge base (KB) optimization by independently optimizing each document within the KB using a separate PROMPTAGENT instance. Unlike STACKFEED, PROMPTAGENT-E operates as a collection of document-wise Independent Actor-Critic models (Foerster et al., 2017).

A.3.1 Algorithm Description

The PROMPTAGENT-E algorithm proceeds as follows:

Dataset	Train	Eval	Test	Documents
Pony	31	32	45	601
ScipyM	22	22	98	3921
TensorflowM	9	9	26	5859
CLEvor News	30	30	60	138

Table 4: Dataset Statistics

1. **Initial Evaluation:** Given a training set of queries, we first run the current KB to obtain retrievals, generations, and expert feedback for these generations. The same is done for the validation set.

2. **Document-Level Dataset Creation:** The training set is then segmented into document-level training sets. Each document-level set comprises all queries (along with their corresponding retrievals, generations, and expert feedback) for which a specific document was retrieved. Similarly, the validation set is also split into document level validation sets.

3. **Document Selection for Editing:** Given that KBs can be extensive, we restrict editing to documents that were retrieved for at least two queries in the training set. This ensures focusing on more relevant or frequently accessed documents.

4. **Independent Optimization:** A separate PROMPTAGENT instance is then created and executed for each selected document independently. Each document-level PROMPTAGENT instance only accesses the queries, generations, and feedback pertinent to its assigned document.

5. **KB Update:** After determining the optimal node (or prompt) for each document, these optimized nodes are integrated back into the KB to form a new, improved version.

A.4 Dataset

Knowledge Base Editing can be useful for scenarios where the KB is

1. Incomplete: the knowledge bases misses some key artifacts responsible for answering the questions. In the Evor-Pony dataset, the documentation used lacks information on various aspects of the language like partial functions etc.

2. Incorrect: the knowledge base in this case consists of some incorrect knowledge.

. We evaluate STACKFEED on 5 datasets spanning these different settings.

A.4.1 Incomplete Knowledge Base

We adapt *two* code generation datasets from Evor (Su et al., 2024), namely **Evor-Pony**. The dataset consists of LeetCode problems and their solutions in low-resource languages Pony and Ring respectively. Each datapoint is supplemented with a corresponding language documentation, with execution accuracy as the success metric and execution failures as feedback to the system. Given that these languages don't appear prominently in LLM pre-training data, the performance of code generation RAG agents on these datasets depends significantly on the quality of the Knowledge Base. However, given that these languages have smaller communities, their documentation isn't as well maintained and often lacks critical information. . For the purpose of evaluation on these datasets, we split them into train, eval, and test splits as specified in Table 4. To ensure that we have a good representation of failure cases during training, we first execute the RAG pipeline on the entire dataset and divide the failures at random in a 1:1:2 ratio for train, eval, and test respectively. All the datapoints with successful execution matches are put in the test split. We use the compiler feedback from the executions as the expert feedback to the STACKFEED system.

A.4.2 Incorrect Knowledge Base

For evaluating under this setting, we leverage the **Evor-ScipyM** and **Evor-TensorflowM** datasets from Evor and the **CLARK-news** dataset from Erase (Li et al., 2025a). The Evor datasets consist of data science problems sourced from the DS-1000 dataset (Lai et al., 2022), which are to be solved by artificially perturbed versions of scipy and tensorflow libraries respectively, while referring to the original unperturbed documentation. Similar to Pony and Ring, we use the execution accuracy on a test bench as a success metric and use compiler outcomes as expert feedback. We also follow a similar approach for data splitting.

While fact retrieval is one of the most popular use cases of RAG systems, evolving nature of information requires us to keep the knowledge bases up to date. To simulate these dynamic factual knowledge updates we use the CLARK-news dataset from Erase (Li et al., 2025a) which contains questions and their respective answers extracted from Wikidata at different timestamps. Each timestamp is characterized by a set of articles that were added in the data at that time. For our evaluation, we pool all the questions whose answers changed for the *first* time at a given timestamp and split them across train, eval and test splits in a 1:1:2 ratio (Table 4).

A.5 Example of Automatic Edits in Evor-Scipy

This is a case in which both PromptAgent-E and STACKFEED opt to append sections to the end of a document about *sparse matrices* in the modified version of SciPy provided in the Evor Dataset.

Demonstrably, the baseline edits add unnecessary information from the *newScience.algorithm* and *newScience.Distribution* modules to the document. The document is about *sparse matrices* so the addition of information about *lineSearch* and *norm_ppf* is not appropriate for this document and it is causing the document to become incoherent

On the other hand, the edits made by STACKFEED are relevant to sparse matrices and keep the document coherent.

Listing 1: PromptAgent-E Edits (Only showing the added sections)

```
+# newScience.algorithm Module

+## Functions

* lineSearch(func, grad, initial_point,
    ↪ direction, **kwargs)
    Perform a line search to find the
        ↪ step size (alpha)
    that satisfies the strong Wolfe
        ↪ conditions.

    Parameters:
    - func : callable
        The objective function to be
            ↪ minimized.
    - grad : callable
        The gradient of the objective
            ↪ function.
    - initial_point : ndarray
        The starting point for the line
            ↪ search.
        Must be provided as a NumPy
            ↪ array.
  . . .

# newScience.distribution Module

## Functions

* norm_ppf(p_values)
Convert p-values to z-scores using the
    ↪ standard normal
distribution. This function is also
    ↪ known as the inverse
of the cumulative distribution function
    ↪ (CDF) for the
standard normal distribution.

Parameters:
- p_values : ndarray
    An array of p-values to be converted
        ↪ .

Returns:
- z_scores : ndarray
    The corresponding z-scores for the
        ↪ given p-values.

Example:
```python
import newScience.distribution as nsd
import numpy as np
p_values = np.array([0.05, 0.025,
    ↪ 0.975])
z_scores = nsd.norm_ppf(p_values)
print(z_scores)
```

Listing 2: STACKFEED Edits (Only showing added sections)

```
### Custom Sparse Matrix Classes

When working with custom implementations
    ↪  of sparse matrices,
it is crucial to understand that the
    ↪ attributes and methods
described in this documentation are
    ↪ specific to the
`scipy.sparse.csr_matrix` class.
Custom sparse matrix classes may not
    ↪ have the same interface or
support the same operations.

#### Disclaimer on Custom
    ↪ Implementations

Developers should not assume that custom
    ↪  sparse matrix classes
will have attributes like 'nnz' or
    ↪ methods such as 'nonzero'.
Instead, they should refer to the
    ↪ documentation or
implementation of the custom class to
    ↪ understand the available
attributes and methods.

##### Example of Incorrect Assumption

```python
# Incorrect assumption that a custom
    ↪ SparseMatrix class has
'nnz' attribute
sparse_matrix = SparseMatrix(...)  #
    ↪ Custom sparse matrix class
if sparse_matrix.nnz == 0:
    print("The sparse matrix is empty.")
# This may raise AttributeError if 'nnz'
    ↪  is not an attribute of
SparseMatrix
Example of Correct Usage
python
Copy
Edit
# Correct approach using methods
    ↪ supported by the custom class
sparse_matrix = SparseMatrix(...)  #
    ↪ Custom sparse matrix class
if sparse_matrix.count_nonzero() == 0:
    print("The sparse matrix is empty.")
# Alternatively, convert to a dense
    ↪ array and use numpy's
count_nonzero
dense_array = sparse_matrix.toarray()
if np.count_nonzero(dense_array) == 0:
    print("The sparse matrix is empty.")
```

A.6 Prompts used in STACKFEED

```
"""
There exists a Language Model based
  software named CodeRAG that
  automatically does the following task
  for a developer:
{task} - {task_desc}

CodeRAG uses a knowledge base to perform
   this task:
{kb_desc}
```

```
A developer used CodeRAG to perform the task on multiple files, and CodeRAG made some errors on them.

Here is one knowledge base file that was involved in these errors:
"""
for i, file in enumerate(kb_files):
    prompt += f"""
File {i+1}:
id: {file['id']}
content: \n<file>\n{file['content']}\n</file>\n"""
    if "special_notes" in file and file["special_notes"] != "":
        prompt += f"""\nspecial_notes: {file['special_notes']}"""

"""
The following are the reflections on the errors made by CodeRAG:
{reflections_str}

The reflections show the relationship of the file with the errors made by CodeRAG.
If the file is named "None," it means the information about the error on which the reflection is based does not fully fit any knowledge base file.

Your task is to use the reflections on the errors made by CodeRAG and provide a generalization on the issues with the file and how it can be improved to prevent the errors.

You should mention common issues found in the reflections and provide a plan for improving the knowledge base files to prevent future errors. Use the reflections to suggest additions or changes in the file, explaining what new content should be added to prevent errors. Before suggesting your plan, give context on the errors using code snippets and other relevant information from the reflections.

You have a scratchpad to reason and plan your generalization. Your scratchpad is for your use only and will not be shared with anyone else.
The scratchpad is represented by the <scratchpad></scratchpad> tags.

Your generalization should follow this format:
<scratchpad>
The contents of the scratchpad
</scratchpad>
<generalization>
Your generalization for this file
</generalization>

You must provide the filled-out scratchpad and generalization in the above format.

General guidelines:
1. Carefully analyze the reflections to understand the errors CodeRAG is making.
2. "None" is a special file, representing that to fix the error, the information should be in a new file.
"""
```

Listing 3: Generalization Stage Prompt

```
"""
Your task is to reflect upon the errors made by CodeRAG based on the user feedback and provide a reflection on the role of the knowledge base files in the making of those errors.

Your reflection should be very specific to the knowledge base files as these reflections will be used to improve the knowledge base files to prevent such errors in the future.

There may be other causes for the error, but you should only focus on whether the knowledge base files could have prevented the error.

You should also provide a way for improving the knowledge base files to prevent the error from happening again.

You should try and see if there is any error in the information provided by the knowledge base or if the knowledge base is missing some information that could have prevented the error.

You also have to figure out if the file should be edited or not. That you do through the needs_editing flag.

You have a scratchpad in which you can reason and plan your reflection. Your scratchpad is for your use only and will not be shared with anyone else. This scratchpad is represented by the <scratchpad> tags.

Your output should be in the following format:

<scratchpad>
The contents of the scratchpad
</scratchpad>

<reflection>
<File 1>
File: Name of the first file
needs_editing: True/False
Reflection: The reflection for this file
</File 1>

<File 2>
File: Name of the second file
needs_editing: True/False
Reflection: The reflection for this file
</File 2>
```

```
...
</reflection>

You have to provide the filled-out
  scratchpad and the reflection in the
  above-described formats. You have to
  reflect on all the files that were
  extracted for the code file.

Here are some general guidelines to
  follow:
1. You should first analyze the question
  , the test bench, the feedback, and
  the output to understand the error
  made by CodeRAG.
2. Then you should carefully analyze the
   knowledge base files to see if the
  theme and the contents of any
  knowledge base file are relevant to
  the error. Particularly, you should
  look out for files that have a factual
   error related to the error or are
  missing some information which should
  have been in the file according to the
   theme of the file.
     a. Read the content of the file and
       understand the theme of the file.
       The theme of this file is of
       course based on the file ID and
       the content of the file but you
       should also consider its
       positioning in the knowledge base.
        That means you should consider
       the other files that were
       extracted for the code file and
       see how this file fits in with
       them. For example, if the file is
       a very basic general guide to the
       task with other files providing
       more detailed information, then it
        would make sense for this file to
        not have detailed information
       about specific cases.
     b. See if the file has any
       information related to the error.
       Check for relevant keywords and
       how the file might have biased the
        language model to make the error.
     c. If the file has information
       related to the error, see if the
       information is correct and
       complete. If the information is
       incorrect or incomplete, the file
       is responsible for the error.
     d. If it doesn't have information
       related to the error, check if it
       makes sense for the file to have
       information related to the error.
       If it doesn't make sense, the file
        is not responsible for the error.
        When deciding this, check whether
        the information would be better
       suited in any of the other
       knowledge base files. If the
       missing info fits better in
       another file, then deem this file
       to not be responsible for the
       error as the missing content can
       be better placed in the other file
       .
     e. If the file is responsible for
       the error, explain the error in
       your reflection and set the
       needs_editing flag to True. And if
        the file is not responsible for
       the error, set the needs_editing
       flag to False.
3. If none of the files have any error
  or if you think the content for the
  error should be in a new file, put a
  file with the name "None" in your
  reflection and for its reflection,
  describe the error and mention why it
  is not due to the knowledge base files
  . For the "None" file, the
  needs_editing flag should always be
  set to True. The "None" file should be
   placed as File n+1 where n is the
  number of files extracted for the code
   file.
4. Choose the least number of files for
  editing, we want to change as few
  files as we can for any error. For
  example, if we have 5 knowledge base
  files, unless very extreme cases, we
  wouldn't want to set the needs_editing
   flag as True on more than 2 files.
  Figure out what the most relevant
  files for the error are and focus on
  them.
5. When you choose to edit multiple
  files, you should make sure that their
   involvements in the error are
  distinct and not overlapping. If they
  are overlapping, think about whether
  changing one file would be enough to
  fix the error.
"""
```

Listing 4: Selection Stage Prompt

```
"""
There exists a Language Model based
  software named CodeRAG that
  automatically does the following task
  for a developer:
{test_bench_code}

The test bench code gives a code where a
   function must be inserted and then it
   is tested with some

test cases.

CodeRAG then outputted the following
  code to answer the question:

if task_desc != "":
    prompt += f"""
{task} - {task_desc}
"""
else:
    prompt += f"""
{task}
"""

prompt += f"""
```

```
The developer used CodeRAG for a
  question. The question is as follows:
{query}

In the question, the developer provided
  the following test bench code:
{test_bench_code}

The test bench code gives a code where a
   function must be inserted and then it
   is

tested with some test cases.

CodeRAG then outputted the following
  code to answer the question:
{output_code}

Based on the above output, the developer
   gave the following feedback to
  CodeRAG:
{feedback}

CodeRAG uses a knowledge base to do this
   task
{kb_desc}

The following files were extracted for
  this particular code file (the content
   of

each file is surrounded in <file></file>
   tags):
"""
for i, instruction in enumerate(
  instructions):
    prompt += f"""
File {i+1}:
id: "{instruction['id']}"
content: \n<file>\n{instruction['content
  ']}\n</file>\n
"""
    if "special_notes" in instruction
      and instruction["special_notes"]
      != "":
        prompt += f"""\nspecial_notes: {
          instruction['special_notes
          ']}"""

prompt += """
Your task is to reflect upon the errors
  made by CodeRAG based on the user
  feedback.
You have to explain in detail the error
  made by CodeRAG. The reflection should
   be

very specific to the question, the
  output code and the feedback.
You should start by explaining the
  question that CodeRAG was asked to
  solve before talking about the error.

Your reflection should have relevant
  code snippets from the output

code which have errors and what should
  be done to fix them.
You should also add a small code example
   to demonstrate the error and
  potential methods to fix it.

You can talk about multiple different
  methods here to address the error.

You have a scratchpad in which you can
  reason and plan your reflection.

Your scratchpad is for your use only and
   will not be shared with anyone else.

Your reflection should be in the
  following format:
<scratchpad>
The contents of the scratchpad
</scratchpad>
<reflection>
Your reflection
</reflection>
"""
"""
```

Listing 5: Reflection Stage Prompt