AGENTIC FRAMEWORK FOR COMPLEX SOFTWARE ENGINEERING QUERY ANSWERING

Team 53
team53@gmail.com

ABSTRACT

We introduce the *Software Engineer Bot*, an Agentic Retrieval-Augmented Generation (RAG) framework designed for handling complex software engineering queries. The architecture employs query decomposition via Qwen and assigns specialized agents for sub-query resolution, including Code Generation, Documentation, and Code Correction Agents. Post-generation, automatic invocations ensure error-free outputs and thorough documentation. Efficiency is enhanced through a Query Complexity Classifier and cognitive-inspired short-term and core memory systems. Using the Pathway framework for data management, the system effectively addresses challenges like accurate retrieval, secure code execution, and multi-hop reasoning. Evaluation on a curated multi-domain dataset highlights its potential for intelligent and adaptive problem-solving in software engineering.

1 INTRODUCTION

Recent advancements in Large Language Models (LLMs) have significantly enhanced their ability to generate coherent and contextually relevant text. However, despite these improvements, LLMs still encounter limitations in handling tasks requiring precise, up-to-date, or specialized knowledge beyond their training data. This challenge is particularly evident in software engineering, where high factual accuracy, domain-specific knowledge, and dynamic decision-making are critical. Retrieval-Augmented Generation (RAG) techniques have emerged to address these limitations by enabling real-time retrieval of relevant information from external sources, thus improving factual accuracy and adaptability to domain-specific queries.

Building on the strengths of RAG, the **Software Engineer Bot** introduces an *agentic* approach, where autonomous agents specialize in distinct tasks, enabling dynamic selection and execution of retrieval strategies. Each agent is designed for specific responsibilities: generating and correcting code, creating documentation, managing organizational information, and providing classical software engineering knowledge. These agents are integrated into a modular, scalable architecture that efficiently processes complex multi-hop queries, emphasizing structured reasoning and step-by-step action planning.

The architecture employs a query breakdown mechanism using Qwen to decompose complex tasks into manageable sub-queries. A similarity-based selection mechanism dynamically assigns each sub-query to the most relevant agent. Short-term memory ensures continuity within a session by storing query-answer pairs, while core memory retains organization-specific information for long-term use, facilitating adaptive and context-aware responses. We employ two forms of **guardrailing**, First our own custom guardrailing where Security and correctness are prioritized through tools such as the Code Jailbreak Check Tool, which enforces safe code execution in a secure sandbox, and NeMo Guardrails, which ensure compliance with predefined safety constraints.

Efficiency is further enhanced using the Pathway framework for vector database management, allowing efficient retrieval and context filtering. A Query Complexity Classifier Tool optimizes resource usage by selecting appropriate contexts based on the complexity of each query. Evaluation of the architecture on a curated dataset of 180 queries across varying difficulty levels demonstrates its capability to handle diverse, domain-specific, and multi-faceted software engineering tasks. Responses were human-evaluated to account for logical reasoning and the multi-solution nature of software engineering problems, highlighting the architecture's robustness and reliability.

This report presents the Software Engineer Bot as a comprehensive solution for leveraging Agentic RAG in software engineering, combining structured reasoning, robust security measures, and efficient memory utilization. The architecture establishes a foundation for intelligent, knowledge-aware systems capable of addressing complex, context-sensitive queries in real-time, paving the way for advancements in autonomous decision-making and domain-specific knowledge management.

2 Related Works

Recent advancements in retrieval-augmented generation (RAG) and reasoning frameworks for language models (LLMs) have been rapidly evolving, particularly for tasks requiring reasoning, decision-making, and context management. One notable framework, **ReAct: Synergizing Reasoning and Acting in Language Models** (Yao et al. (2022)), enables LLMs to combine reasoning and action by generating reasoning traces alongside task-specific actions, allowing models to interact with external sources, such as knowledge bases, to gather additional information when necessary. This synergy not only helps mitigate common issues like hallucination but also provides improved human interpretability and trustworthiness by generating clear task-solving trajectories.

Another important framework, **Adaptive RAG**(Jeong et al. (2024)), tackles the query classification and retrieval process by incorporating a lightweight language model that assesses whether a query requires retrieval. This approach efficiently handles the trade-off between retrieval and reasoning, ensuring that resources are utilized effectively for a given task. Similarly, **Corrective RAG** (Yan et al. (2024)) enhances this by introducing a fallback mechanism, which allows for web searches or other forms of external retrieval when the quality of the retrieved documents is inadequate. This approach also focuses on filtering out irrelevant context, thus limiting token usage and preventing hallucinations due to unnecessary context.

The **Self-RAG** framework(Asai et al. (2023)), on the other hand, extends RAG by introducing self-reflection through the use of special tokens that help the model decide when documents should be retrieved, and more importantly, when they are relevant. Although this method is promising, its reliance on large, costly models like GPT-4 for generating training data for the special tokens may limit its accessibility.

Other works such as **Chain of Thought (CoT)** (Wei et al. (2022)) and **Tree of Thoughts (ToT)** (Yao et al. (2024)) explore alternative reasoning strategies. CoT focuses on reasoning through chains of thought, but lacks the capacity to perform actions, limiting its ability to gather useful context during a query. ToT improves upon this by enabling exploration over coherent units of text, providing better strategic decision-making for tasks that require planning or multi-step reasoning.

The **Graph of Thoughts** framework (Besta et al. (2024)) aims to model reasoning in the form of a directed acyclic graph, but it suffers from computational overhead due to multiple LLM calls.

Additionally, **M-RAG**(Wang et al. (2024)) applies reinforcement learning to select relevant partitions for answering queries, while **Struct RAG**(Li et al. (2024)) works on identifying optimal structural representations of data, though both are limited by their reliance on closed-source LLMs.

In the context of task execution, **Tape Agents** provide a structured, granular log of agent interactions, facilitating all stages of the LLM agent lifecycle, including memory management. The inclusion of a **TAPE memory** in our framework will be pivotal for session continuity, as it will allow the system to remember previous actions and decisions, improving efficiency and adaptability across multiple tasks.**MetaGPT**(Hong et al. (2023)) introduces a meta-programming approach designed for multi-agent collaborative frameworks, which enables flexible and dynamic coordination among agents in various tasks.

Lastly, **MARCO** (Shrimal et al. (2024)) and **NeMo Guardrails**(Rebedea et al. (2023)) address critical challenges in LLM-based systems by incorporating robust guardrails to steer model behavior, validate outputs, and recover from errors.NeMo Guardrails, in particular, offer an open-source toolkit to programmatically control LLM outputs, ensuring that responses adhere to predefined safety and quality standards.

3 Challenges in Existing RAG Systems and Proposed Solutions

Current Retrieval-Augmented Generation (RAG) systems face significant challenges in achieving accurate, efficient, and secure outputs, particularly in domains like software engineering that demand high precision and contextual understanding. Extracting precise answers from retrieved content is often problematic; our architecture addresses this by incorporating a Query Complexity Classifier Tool, which minimizes unnecessary retrievals and optimizes resource allocation. Scalability challenges with data ingestion are mitigated through the Pathway framework, which ensures efficient vector database creation and retrieval, alongside template-specific chunking for improved context filtering.

To address the critical need for secure code execution, our system employs a Code Jailbreak Check Tool, a robust sandboxing solution that enforces stringent security measures, preventing malicious or unintended code execution. Missing knowledge, a common issue in LLM systems, is handled via fallback mechanisms like web search in the Software Engineering Knowledge Agent, ensuring comprehensive context for complex queries. Additionally, NeMo Guardrails are integrated into specific agents, enhancing output reliability by filtering unsafe or irrelevant responses.

Inefficient memory utilization, a key bottleneck in conventional RAG systems, is addressed by introducing a dual-memory structure. Short-term memory ensures session continuity by storing relevant query-answer pairs, while core memory facilitates long-term retention of organization-specific information. These innovations collectively enable the architecture to deliver accurate, scalable, and secure solutions for software engineering tasks.

4 Target Use Case for the Software Engineer Bot

The proposed *Software Engineer Bot* is designed to address complex, multi-domain software engineering queries, combining dynamic retrieval capabilities, robust security measures, and autonomous decision-making. This architecture supports tasks like code generation, debugging, documentation creation, and knowledge retrieval in software development environments.

By leveraging specialized agents, the system provides end-to-end solutions for diverse challenges in software engineering. A key use case involves handling multi-hop queries across domains such as language specifications, API references, and codebase documentation. The integration of NeMo Guardrails and the Code Jailbreak Check Tool ensures safe and contextually relevant responses, even in scenarios requiring code execution or sensitive data handling.

4.1 Agents Incorporated in the Software Engineer Bot

The architecture employs the following specialized agents, each tailored to specific functionalities:

- **Code Generation Agent:** Automatically generates code snippets based on input prompts, leveraging contextual information from the retrieval pipeline. It ensures adherence to best practices and guidelines, producing syntactically and semantically correct outputs.
- **Code Correction Agent:** Automatically called after code generation, this agent validates and refines the generated code. It checks for errors, optimizes performance, and aligns the output with organizational standards using data from short-term and core memory.
- **Documentation Agent:** Also automatically invoked post-code generation, this agent creates comprehensive documentation for generated code. It retrieves relevant information from API documentation and language specifications, aiding in code usability and maintainability.
- **HR Manager Agent:** Provides organizational information, ensuring queries related to employee data or policies are handled securely. It incorporates a Query Filtering Tool to manage sensitive information in compliance with data governance standards.
- **Software Engineering Knowledge Agent:** Addresses sub-queries requiring classical software engineering knowledge or specialized domain expertise. It uses fallback mechanisms like web search to enhance context completeness for queries outside its primary scope.

These agents operate cohesively within a structured pipeline, utilizing advanced tools such as the Query Complexity Classifier and the Pathway framework for efficient query resolution. Together, they form a robust, adaptive solution tailored to software engineering challenges, highlighting the versatility and reliability of the Software Engineer Bot architecture.

5 Proposed Solution Architecture

In order to evaluate the efficacy of open-source large language models (LLMs) in decomposing complex queries into simpler, single-hop queries, we benchmarked the models on the **FRAMES Dataset** (Krishna et al. (2024)). Given the relatively small size of the dataset, consisting of only 824 queries, we manually assessed the results. The evaluation revealed that open-source LLMs are indeed capable of effectively breaking down complex queries.

Following the success of open-source LLMs in query decomposition, we propose an agent-based approach for query resolution. This involves associating LLMs with agents via a similarity search. Specifically, we plan to index embeddings of agent descriptions and assign the most suitable agent to a query based on the highest similarity score. Once an agent is selected, it is equipped with various specialized tools, each of which is described in the following section.

5.1 Tools

Code Execution Tool: This tool enables agents to execute code securely and automatically. It is particularly essential for the **Code Generation Agent** and **Code Correction Agent**, allowing them to validate generated code by running it within a controlled environment. This ensures correctness and minimizes hallucinations in computational tasks, providing reliable results for mathematical, logical, and coding-related queries.

Code Jailbreak Check Tool: A robust sandboxing environment, leveraging containerization technologies like Docker, is utilized for secure code execution. This tool enforces stringent resource limits, network isolation, and file system access controls during execution. Comprehensive input/output validation, memory bounds checking, and runtime monitoring are performed to detect and prevent potential malicious behavior or attempts to escape the container. This tool is a critical component of the **Code Generation Agent** and **Code Correction Agent**, ensuring safe and reliable execution of user-generated or system-generated code. It also contributes to enforcing security guardrails throughout the code-related tasks.

Query Complexity Classifier Tool: This tool assesses the complexity of incoming queries and determines the appropriate retrieval strategy. It categorizes queries into three types: **no retrieval**, **single retrieval**, or **multi-retrieval** using a trained T5 model. For **no retrieval**, the query is handled directly by the LLM without external data. For **single retrieval**, a single chunk of relevant information is fetched, while **multi-retrieval** involves iterative fetching of multiple chunks until the query is resolved. This tool is universal and used across all agents to optimize resource allocation and ensure efficient query resolution.

Query Filter and Web Search Tool: This tool extracts the most relevant context from documents retrieved by the RAG Tool or the web. It scores each document using a T5 sequence regressor, classifying them as **Correct**, **Ambiguous**, or **Incorrect**:

- **Correct:** Chunks with scores exceeding the upper threshold are passed to the LLM as refined knowledge.
- **Incorrect:** Documents with scores below the lower threshold trigger a web search for retrieving alternative context.
- **Ambiguous:** Documents falling between thresholds are handled using a hybrid approach, combining retrieval refinement and web search strategies.

This tool is critical for enhancing the accuracy of **Software Engineering Knowledge Agent** , particularly when retrieving or refining domain-specific knowledge or resolving ambiguous queries.

RAG Tool: A foundational component for all agents, this tool performs standard retrieval using a **FAISS-based similarity search** to fetch relevant chunks from their respective databases. Each

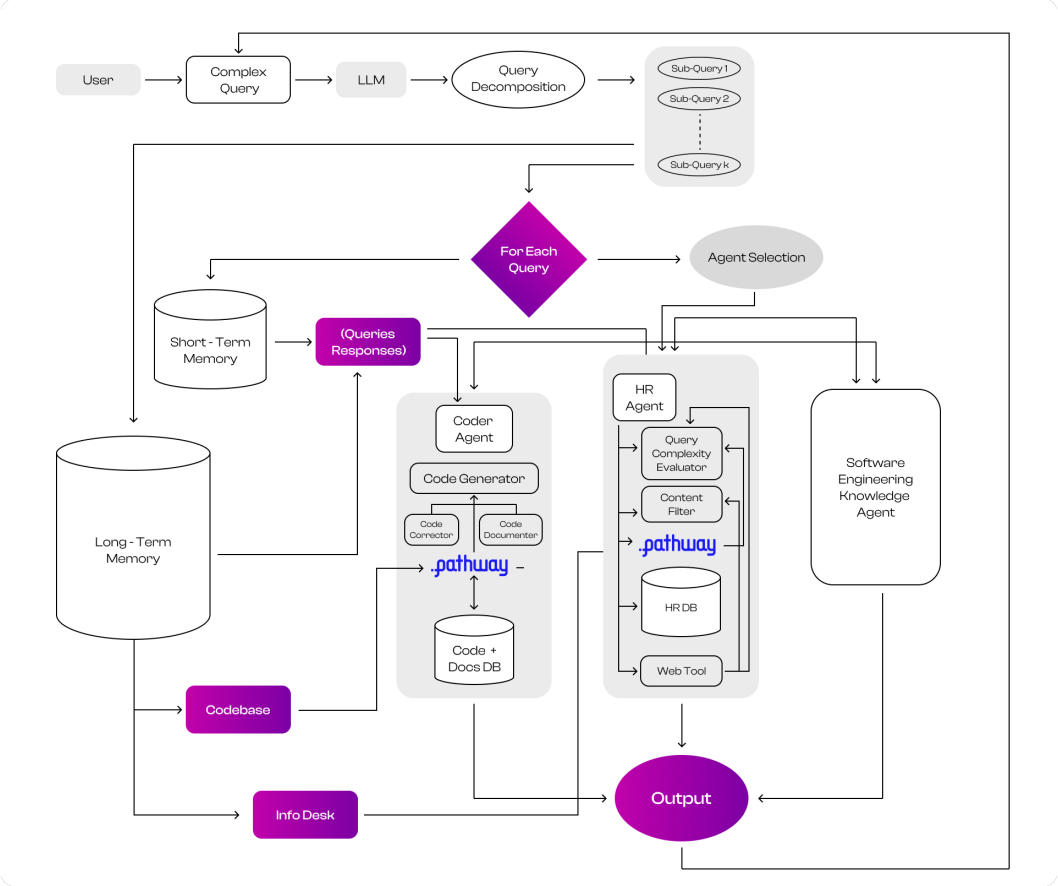

Figure 1: Proposed System architecture

agent, including the **Code Generation Agent**, **HR Manager Agent**, and **Software Engineering Knowledge Agent**, leverages the RAG Tool to access domain-specific or task-specific information.

5.2 Agentic Pipeline

The agentic pipeline integrates a series of structured and interdependent processes to handle complex queries efficiently and accurately. It encompasses query breakdown, agent assignment, complexity analysis, retrieval filtering, and advanced memory management. Additionally, robust guardrailing mechanisms ensure the safety and reliability of outputs. Each step contributes to creating a scalable, adaptable, and secure solution framework, as depicted in Figure 1.

5.2.1 Query Breakdown

The pipeline begins with query decomposition to handle complex queries systematically. Using the Qwen model, the input query is divided into sequential sub-queries, following a directed acyclic graph (DAG). This step ensures that each component of the query is addressed methodically, enabling the system to solve multi-step problems in a logical and structured manner. Query decomposition is critical for tasks requiring iterative reasoning or multiple sources of information.

5.2.2 Agent Assignment

After the query is decomposed, each sub-query is assigned to an appropriate agent using similarity search. An index of agent embeddings, representing the descriptions and capabilities of all available agents, is leveraged for this process. The agent with the highest similarity score to the sub-query is chosen, ensuring that each task is handled by the most relevant specialized agent. This approach

ensures precise responses while maintaining efficiency through targeted task allocation. This agent assignment helps add any number of agents as you want easily.

5.2.3 Query Complexity Analysis

Once agents are assigned, the **Query Complexity Classifier Tool** assesses the complexity of each sub-query using a trained T5 model. Based on its evaluation, the sub-query is categorized into one of three classes:

- **No Retrieval:** The query is resolved directly by the LLM without external context.
- **Single Retrieval:** A single chunk of relevant context is fetched from the database.
- **Multi-Retrieval:** Multiple chunks are iteratively retrieved to comprehensively address the query.

This classification ensures efficient resource utilization by tailoring the retrieval strategy to the specific needs of each sub-query.

5.2.4 Memory Management: Short-Term and Long-Term Memory

Memory management in the agentic pipeline is inspired by cognitive processing, utilizing two types of memory to balance immediate context with long-term adaptability.

Short-Term Memory Short-term memory stores query-answer pairs generated during the current session. Unlike methods like ReAct that rely on the entire interaction history, our approach selectively retrieves only the most relevant past sub-query pairs based on similarity scores from a trained cross-encoder. This minimizes token usage while ensuring contextual relevance for subsequent sub-queries, optimizing the agent's response efficiency within a session.We used BgeEmbedder to retreive the relevant information from the short term memory.

Long-Term Memory (Core Memory) Core memory provides persistent storage for query-answer pairs across sessions. This enables the system to retain organization-specific or domain-specific knowledge, adapting to recurring and complex queries over time. Stored information is curated and can be manually updated, ensuring that it remains relevant and precise for organizational use cases.We used BgeEmbedder to retreive the relevant information from the short term memory.

5.2.5 Custom and Programmable Guardrailing Mechanisms

To maintain the safety and reliability of outputs, the agentic pipeline incorporates both custom and programmable guardrails:

Custom Guardrailing Using the Code Jailbreak Check Tool For queries involving code execution, the **Code Jailbreak Check Tool** ensures secure execution within a controlled environment. Leveraging technologies like Docker, the sandbox environment imposes strict resource limits, network isolation, and file system access controls. Additionally, runtime monitoring detects malicious behavior or attempts to escape the sandbox. This ensures that all code is executed safely without risking system integrity.

NeMo Guardrails NeMo Guardrails are integrated at every LLM node in the pipeline. This open-source framework enforces rules governing the output of language models. The guardrails restrict off-topic or harmful content, enforce predefined conversational flows, and maintain the desired tone and style. These programmable safeguards complement custom tools, offering an additional layer of reliability and adherence to organizational policies.

5.3 Leveraging Pathway for Efficient Database Management and Retrieval

To enhance the efficiency and scalability of our agentic framework, we utilized **Pathway's core capabilities** for vector database creation and retrieval. Pathway, a framework known for its robust handling of dataflows and dynamic processing, provided an ideal foundation for implementing our innovative solution.

Architecture	Easy	Medium	Hard	Total
CodeLlama (Zero-Shot)	20	10	20	16.66
Codestral (Zero-Shot)	53.33	60	40	56.67
CodeLlama (RAG)	40	20	20	30
Codestral (RAG)	66.67	60	40	63.33
ME-QT5 (Proposed)	**73.33**	**80**	**80**	**76.67**

Table 1: Performance(Accuracy in percentage) of various architectures on curated dataset.

5.3.1 Database Creation with Pathway

Pathway's dynamic graph architecture allowed us to design highly modular and flexible databases tailored to the requirements of each agent in the framework. The process involved:

- **Dynamic Schema Definition:** Pathway's ability to define custom schemas enabled the storage of diverse data types, such as code snippets, API documentation, and natural language descriptions. Each database was aligned with the specific retrieval needs of agents, ensuring optimized storage and access.
- **Real-Time Updates:** The framework's capability to handle streaming data facilitated real-time updates to the database, ensuring that the system remained up-to-date with the latest information, such as newly added documentation or organizational knowledge.
- **Efficient Chunking:** Using Pathway's processing pipelines, we implemented template-specific chunking mechanisms. Each document was chunked based on paragraph embeddings and structural context, enhancing retrieval accuracy and reducing redundancy.

6 Evaluation

To evaluate the performance of our proposed architecture, **ME-QT5**, we curated a dataset of 180 queries across three difficulty levels: Easy, Medium, and Hard. The Easy category includes code generation questions confined to a single software engineering domain, while Medium and Hard categories involve 3 and 4+ domains, respectively. The dataset consists of 90 examples from the Easy category, 60 examples from Medium, and 30 examples from Hard.

Human evaluation was used to assess response correctness, as traditional NLP metrics like BLEU and ROUGE fail to capture the logical reasoning necessary for code generation and are unable to generalize where queries may have multiple correct answers. The results of **ME-QT5** were compared with other systems, including **CodeLlama (Zero-Shot)**, **Codestral (Zero-Shot)**, and their Retrieval-Augmented Generation (RAG)-based variants whose database is initialized with CodeSearchNet. The metric show is percentage accuracy of number of correct responses.

Table 1 summarizes the performance of these systems across all difficulty levels.

The results show that **ME-QT5** consistently outperforms other architectures, especially in Medium and Hard categories, demonstrating its ability to handle complex multi-hop queries involving multiple software engineering domains.

7 Future Work and Scope for Improvement

While the current framework effectively addresses complex query handling and response generation by leveraging query complexity classifiers and adaptive retrieval, several areas offer promising directions for future enhancement. Furthermore, the database chunk selection process, while currently relying on similarity-based retrieval, could benefit from reinforcement learning (RL) integration. RL agents could be trained to optimize the selection of chunks based on reward mechanisms tied to query resolution efficiency, context relevance, and answer accuracy. For instance, M-RAG's use of RL agents to refine retrieval relevance could inspire similar adaptations in our framework, allowing for more intelligent chunk selection that better aligns with the specific nuances of each query (Wang et al. (2024)). This approach would also enable dynamic adaptation of chunk selection strategies

based on historical user interactions, query types, or domain-specific needs, improving the framework's responsiveness and adaptability. Additionally, exploring novel chunking methodologies, such as adaptive chunk granularity based on query complexity or hierarchical chunking models, could further streamline retrieval and reduce computational overhead. By dynamically adjusting the granularity of retrieved content, the framework could fine-tune response precision, reducing noise while retaining essential context. This approach could be especially beneficial for applications requiring high accuracy in multi-hop retrieval tasks, where irrelevant context must be minimized. In summary, these future directions—including token optimization, RL-based chunk selection, adaptive chunking for advancing the framework's efficiency, adaptability, and domain applicability.

REFERENCES

Akari Asai, Zeqiu Wu, Yizhong Wang, Avirup Sil, and Hannaneh Hajishirzi. Self-rag: Learning to retrieve, generate, and critique through self-reflection. *arXiv preprint arXiv:2310.11511*, 2023.

Maciej Besta, Nils Blach, Ales Kubicek, Robert Gerstenberger, Michal Podstawski, Lukas Gianinazzi, Joanna Gajda, Tomasz Lehmann, Hubert Niewiadomski, Piotr Nyczyk, et al. Graph of thoughts: Solving elaborate problems with large language models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pp. 17682–17690, 2024.

Sirui Hong, Xiawu Zheng, Jonathan Chen, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, et al. Metagpt: Meta programming for multi-agent collaborative framework. *arXiv preprint arXiv:2308.00352*, 2023.

Soyeong Jeong, Jinheon Baek, Sukmin Cho, Sung Ju Hwang, and Jong C Park. Adaptive-rag: Learning to adapt retrieval-augmented large language models through question complexity. *arXiv preprint arXiv:2403.14403*, 2024.

Satyapriya Krishna, Kalpesh Krishna, Anhad Mohananey, Steven Schwarcz, Adam Stambler, Shyam Upadhyay, and Manaal Faruqui. Fact, fetch, and reason: A unified evaluation of retrieval-augmented generation. *arXiv preprint arXiv:2409.12941*, 2024.

Zhuoqun Li, Xuanang Chen, Haiyang Yu, Hongyu Lin, Yaojie Lu, Qiaoyu Tang, Fei Huang, Xianpei Han, Le Sun, and Yongbin Li. Structrag: Boosting knowledge intensive reasoning of llms via inference-time hybrid information structurization. *arXiv preprint arXiv:2410.08815*, 2024.

Traian Rebedea, Razvan Dinu, Makesh Sreedhar, Christopher Parisien, and Jonathan Cohen. Nemo guardrails: A toolkit for controllable and safe llm applications with programmable rails. *arXiv preprint arXiv:2310.10501*, 2023.

Anubhav Shrimal, Stanley Kanagaraj, Kriti Biswas, Swarnalatha Raghuraman, Anish Nediyanchath, Yi Zhang, and Promod Yenigalla. Marco: Multi-agent real-time chat orchestration. *arXiv preprint arXiv:2410.21784*, 2024.

Zheng Wang, Shu Xian Teo, Jieer Ouyang, Yongjun Xu, and Wei Shi. M-rag: Reinforcing large language model performance through retrieval-augmented generation with multiple partitions. *arXiv preprint arXiv:2405.16420*, 2024.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

Shi-Qi Yan, Jia-Chen Gu, Yun Zhu, and Zhen-Hua Ling. Corrective retrieval augmented generation. *arXiv preprint arXiv:2401.15884*, 2024.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2022.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. *Advances in Neural Information Processing Systems*, 36, 2024.