Inter IIT Tech Meet-13.0

End Term Report

Team-79

Contents

1 Approach Outline

1.1 Introduction

Our project focuses on building an Agentic Retrieval-Augmented Generation (RAG) system that integrates autonomous AI agents to enhance the processes of information retrieval and response generation.Unlike native RAG, which relies on static parameters, Agentic RAG allows agents to adapt and reason based on the query and context, improving accuracy and efficiency. Our approach aims to design a system where AI agents can independently analyze query complexity, dynamically retrieve relevant information from indexed databases,critically evaluate the retrieved data,perform actions if required and generate accurate and contextually appropriate responses. By implementing this adaptive framework, the system seeks to improve efficiency, scalability, and reliability in handling a wide range of queries, from simple to highly complex.

1.2 Uniqueness

- **API-Centric Data Retrieval with Scoped Access:** Unlike traditional solutions that query entire databases, our approach leverages the platform's existing API endpoints. By embedding user authentication in the headers of each API call, we ensure data retrieval is both accurate and limited to the user's authorized scope. This provides a highly secure and personalized experience for each user.
- **Intelligent Answer Generation Over Basic Navigation:** Many solutions, like Rufus, act as navigation tools that guide users to relevant pages. While these enhance UX to an extent, they stop short of generating intelligent answers. Our approach goes further by synthesizing intelligently generated answers from the data retrieved via APIs, elevating the user experience significantly.
- **Seamless Integration via OpenAPI Specification:** Our solution is designed for effortless integration with existing platforms by adhering to the OpenAPI specification—a widely accepted standard. This minimizes the need for extensive customization, reducing implementation time and effort while maximizing compatibility and adaptability.
- **Efficiency Through DSPy and Smaller Language Models:** Traditional approaches often rely on large-scale language models like ChatGPT and Gemini to generate structured outputs, which can be resource-intensive and incur higher latency and costs. By integrating DSPy into our system, we enable the use of smaller models, such as Gemma, to produce structured outputs effectively. This integration significantly improves latency and reduces computational overhead, making our solution more cost-effective without compromising on performance or accuracy.

1.3 Use-Case Selection and Novelty

An Agentic RAG that integrates with a platform's existing APIs, exposing the platform's functionality through a chat interface to the end user.

1. The RAG compares the query semantically with the documentation of the available API endpoints to find the most relevant matches to the user query.
2. Once the relevant matches are found, the RAG sends a request to those endpoints to retrieve the required data.
3. A key feature in our concept is maintaining user authentication through all API calls, ensuring:
    - Personalized data retrieval for better quality.
    - Maintenance of the user's scope of data access.

1.4 Solution Overview

Step I: Query Handling

- Rephrasing the query for clarity to ensure it is easily understood and aligns with the system's capabilities.
- Classifying the query to determine if retrieval or additional processing steps are required.
- Decomposing complex queries into simpler sub-queries to facilitate accurate and efficient processing.
- Checking the query for profanity or sensitive data to maintain ethical communication and ensure compliance with privacy and security standards before further processing.

- **Chat History Integration:** To enhance the performance of our Retrieval-Augmented Generation (RAG) pipeline, we integrated chat history into the system. By utilizing previous queries and responses, the system generates more context-aware and accurate responses, thus improving the overall user experience and enabling more seamless interactions.

Step II: Endpoint Selection and Retrieval

- Identifying the most relevant API endpoints by embedding the query and comparing it to endpoint descriptions using similarity scoring.
- Selecting the top-k endpoints and re-ranking them based on relevance to ensure optimal results.
- Ensuring the input data complies with the required schema, typically derived from the *openapi.json* file, before making any API calls.
- Sending requests to the identified endpoints and handling the exchange of information as required.

Step III: Response Generation

- Processing the retrieved information through a network of AI agents to generate a suitable response.
- Performing additional actions, such as code generation or computation, if required by the query, handled by specific agents.

– **Code Generation Agent:** Classifies the query based on whether it requires code generation or simple information retrieval. If code is needed, the agent generates the appropriate code, executes it, and uses the result to generate the response. If no code is needed, the agent proceeds with response generation from the retrieved data.

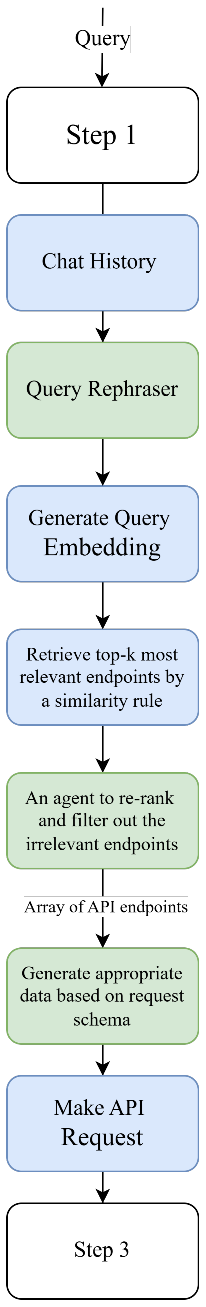
- Verifying the response through guardrails to ensure it contains no inappropriate or sensitive content before final output.

– **Guardrails:** To ensure responses are safe and reliable, we incorporated key guardrails such as filtering irrelevant or sensitive data from API responses and requiring explicit user consent for actions with long-term consequences. These mechanisms uphold privacy, security, and ethical standards while enhancing user trust.

- Presenting the final output, accompanied by follow-up questions for further interaction.

– **Follow-up Queries Agent:** This agent generates potential follow-up queries based on the original query after a response has been provided. This mechanism anticipates user needs, enhancing the conversational flow and encouraging deeper engagement.

These components works in unison to provide efficient, accurate, and safe query processing, resulting in an optimal user experience throughout the interaction.


Query
Step 1
Chat History
Query Rephraser
Generate Query Embedding
Retrieve top-k most relevant endpoints by a similarity rule
An agent to re-rank and filter out the irrelevant endpoints
Array of API endpoints
Generate appropriate data based on request schema
Make API Request
Step 3

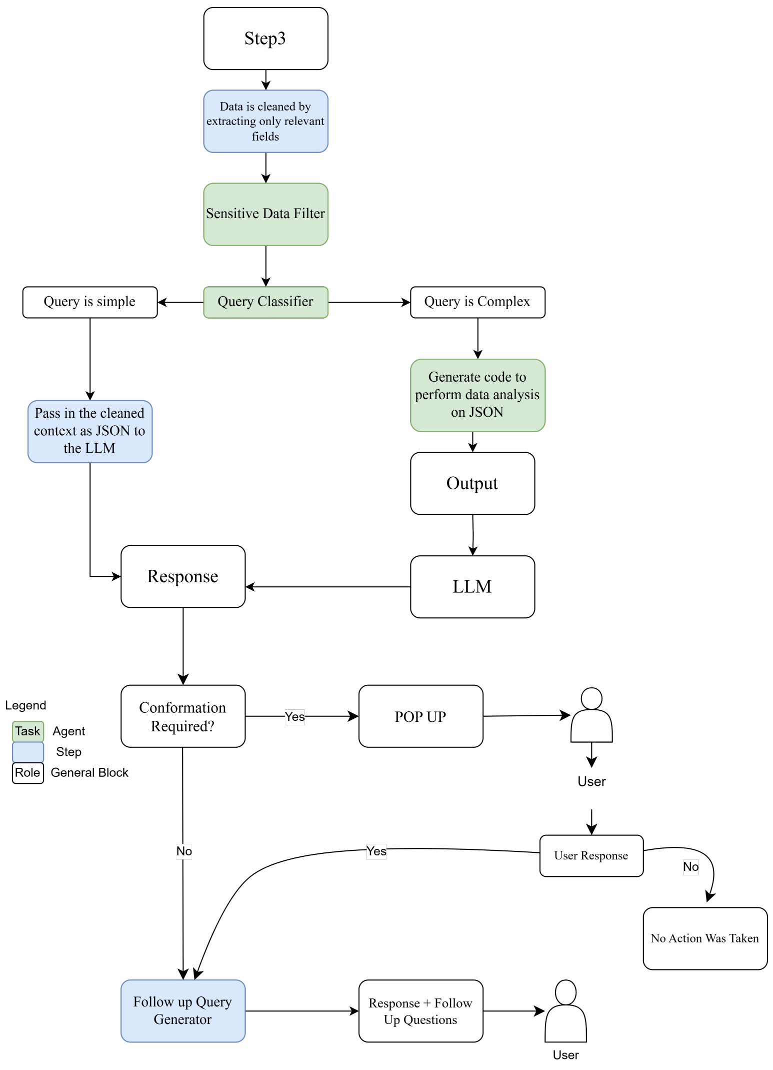
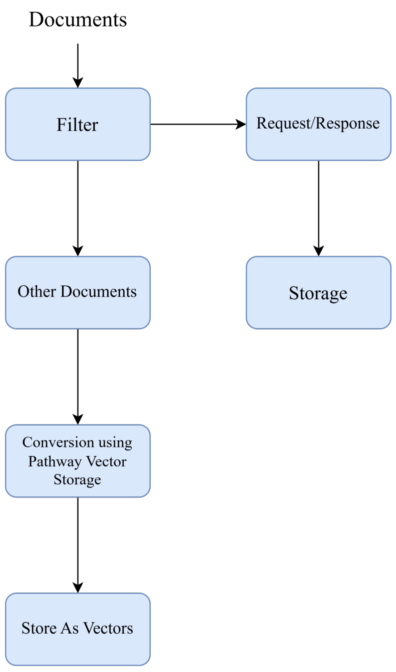


Figure 1:

1.5 System Architecture

1.5.1 Technical Approach

The user's query is first rephrased by an AI agent to align with the semantic structure needed for comparison with API documentation. Then, we enhance our retrieval process by implementing a hybrid search approach that integrates both dense (semantic) and sparse (keyword-based) retrieval methods using Pathway's `DocumentStoreServer`. By creating a hybrid index, we leverage dense embeddings to capture the query's meaning and sparse embeddings for exact keyword matches. During retrieval, a combined scoring system ranks results based on both semantic relevance and keyword accuracy, ensuring that high-relevance documents from either dense or sparse matches appear in the top-k results. Now, the top k endpoints that **might** be useful in answering the query are retrieved. The vector embeddings of the query vector are compared only with the natural language documentations of the available API endpoints and not their request and response schema. Once the top k most relevant endpoints are retrieved, we then retrieve their response schema and and ask an AI agent to filter them and return an array consisting of only the relevant endpoints.



Once the endpoints are successfully retrieved, we retrieve their request schema and use an agent to generate suitable values for the parameters and body required to make the request.

We now face the challenge of cleaning the request response. The request response in most cases will contain fields that are completely irrelevant to answer the query. For example, fields like pagination metadata (e.g., `page`, `total_count`, `next_page_url`) , API versioning or status codes (e.g., `version`, `status`) , session information (e.g., `user_id`, `session_id`, `preferences`), etc. It is important to remove these fields since we need to pass this data as the context to our network of agents later on. These irrelevant fields might create confusion in the context and will consume more tokens than required to answer the query.

Once this data is cleaned, it is passed as context to an agent that is responsible to decide whether this query requires any sort of data analysis or not. If it does, we use a code generation agent to generate an appropriate code that performs data analysis on the information. The output of this code is added to the context and passed in to the final agent responsible for generating a response.

1.6 Implementation

The very first step was to understand how Pathway as a library helps in the case of RAG pipelines. Then, Pathway was locally set up using Docker, and the development environment was configured in VS Code.

We built an API using FastAPI to serve as an interface for the RAG pipeline, taking advantage of FastAPI's Swagger interface, which streamlines the testing process. As we continued refining our RAG pipeline, we implemented a hybrid search approach for the retrieval. We ensemble BM25 and KNN retrieval methods. They brought improvements in accuracy which have been documented in tables below. Our current setup organizes API endpoints by their URL. The folder structure is based on each endpoint's URL. For example, if there's an endpoint for retrieving user data at `/api/users`, the folder might look like this:

```
Storing API Documentation
(Folder Structure With
Request Methods)
|-- api
|   |-- users
|   |    get.txt
|   |    post.txt
```

Inside the `get.txt` and `post.txt` files, we have simple documentation explaining the GET and POST methods for that endpoint. When we fetch the API docs from the "docs" folder (the only one we index with Pathway's `DocumentStoreServer`), we use the file's path in its metadata (provided by Pathway) to figure out the API endpoint's URL. So, the file at `/documentations/api/users/get.txt` would tell us the `/api/users` endpoint corresponds to the GET method.

We then retrieve the request and response schema from a separate folder, which provides detailed information in JSON format about how to structure the request parameters and body, as well as the expected response.
Coming to the implementation of our AI agents, we are implementing AI agents using DSPy. Instead of free-form string prompts, DSPy programs use natural language *signatures* to assign work to the LM. A DSPy *signature* is natural-language typed declaration of a function: a short declarative spec that tells DSPy what a text transformation needs to do (e.g., "consume questions and return answers"), rather than how a specific LM should be prompted to implement that behavior. [8] In practice, DSPy signatures can be expressed with a shorthand notation like `"question->answer"`, or something like `"context, question->answer"` for taking in context as a parameter to answer the question. DSPy generates an optimal prompt based on the signature.
We have two different kinds of agents, **with tools** and **without tools**. **In our use case, we define a tool as an API endpoint that returns data that can be used to complete the parameters and body for a request.** We utilize the ReAct module when the DSPy agent is equipped with a tool, and the Predict module when no tool is available, ensuring optimal speed with accuracy We've also implemented necessary guardrails to ensure safety and reliability. The system filters out sensitive or irrelevant JSON keys from API resonses before passing them as context to the agents fo r response generation. Moreover, the DSPY agent identifies queries with potential long-term or financial consequences and prompts the user for explicit approval before executing any actions. Using these DSPy agents, we generate parameters and body for the requests to be made. After making the requests, we pass in the response as context to our response generation step explained in the technical approach.

2 Analysis and Experimentation

2.1 Results and Metrics

For experimenting and evaluation, we built a test API from scratch mimicking the endpoints that an e-commerce platform would have. The following are the endpoints we considered:

```
GET  "http://127.0.0.1:8000/cart/my"
    This route allows the user to get a list of all of the items in their cart.
POST "http://127.0.0.1:8000/cart/add"
    This route allows the user to add a product to cart.
GET  "http://127.0.0.1:8000/orders/my"
    This route allows the user to get a list of all of their orders.
POST "http://127.0.0.1:8000/orders/place"
    This route allows the user to place order.
GET  "http://127.0.0.1:8000/reviews/my"
    This route allows the user to get a list of all of their reviews.
POST "http://127.0.0.1:8000/reviews/post"
    This route allows the user to post a review on a product.
GET "http://127.0.0.1:8000/products/search"
    This route allows the user to search a product given its name.
```

2.1.1 Sample outputs for retrieval of correct endpoint

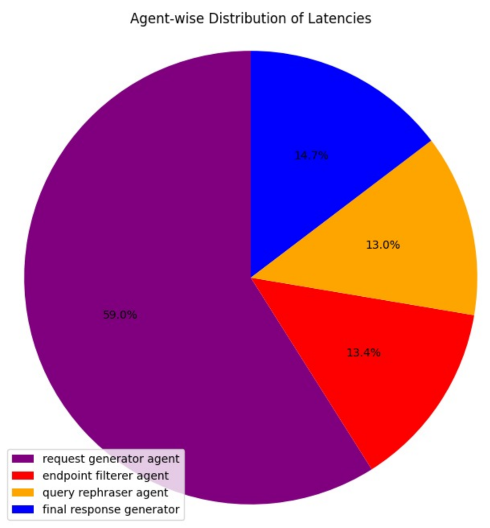
Query: "what stuff do i have in my cart?"
Output (array of relevant endpoints): `["http://127.0.0.1:8000/cart/my"]`

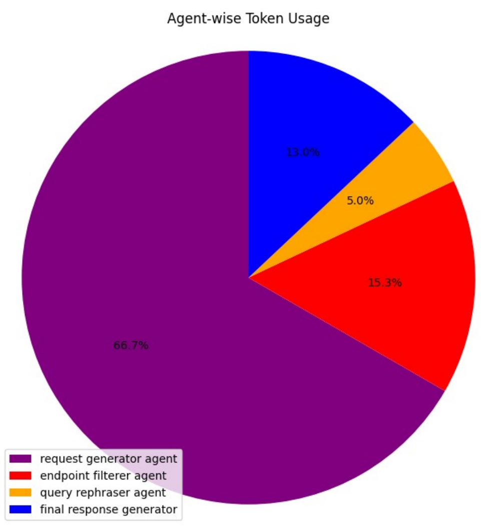
2.1.2 Evaluation Results

We have tested various models paired with different retrieval methods, including hybrid search, BM25, and k-nearest neighbors (KNN), to evaluate their effectiveness across several performance metrics. The evaluation criteria used to assess model performance include the context's **accuracy**, **precision**, **recall**, and **F1 score**:
- **Accuracy** measures the proportion of correct predictions out of all predictions, indicating overall correctness.
- **Precision** is the ratio of correctly retrieved relevant results to the total retrieved results, showing how well the model avoids false positives.
- **Recall** is the ratio of correctly retrieved relevant results to the total actual relevant results, indicating the model's ability to retrieve all relevant data.
- **F1 Score** is the harmonic mean of precision and recall, providing a balanced measure that accounts for both false positives and false negatives.

Method	Context Precision	Context Recall	F1 Score	Accuracy Score
gpt-4o with hybrid search	1.00	0.88	0.94	0.71
gpt-4o-mini with hybrid search	0.93	0.85	0.89	0.71
gemini-pro with hybrid search	0.88	0.67	0.76	0.61
gpt-4o-mini with bm25	0.72	0.64	0.68	0.54
gemini-pro with knn	0.95	0.57	0.72	0.56

Table 1: Performance metrics for correct endpoint retrieval using different methods





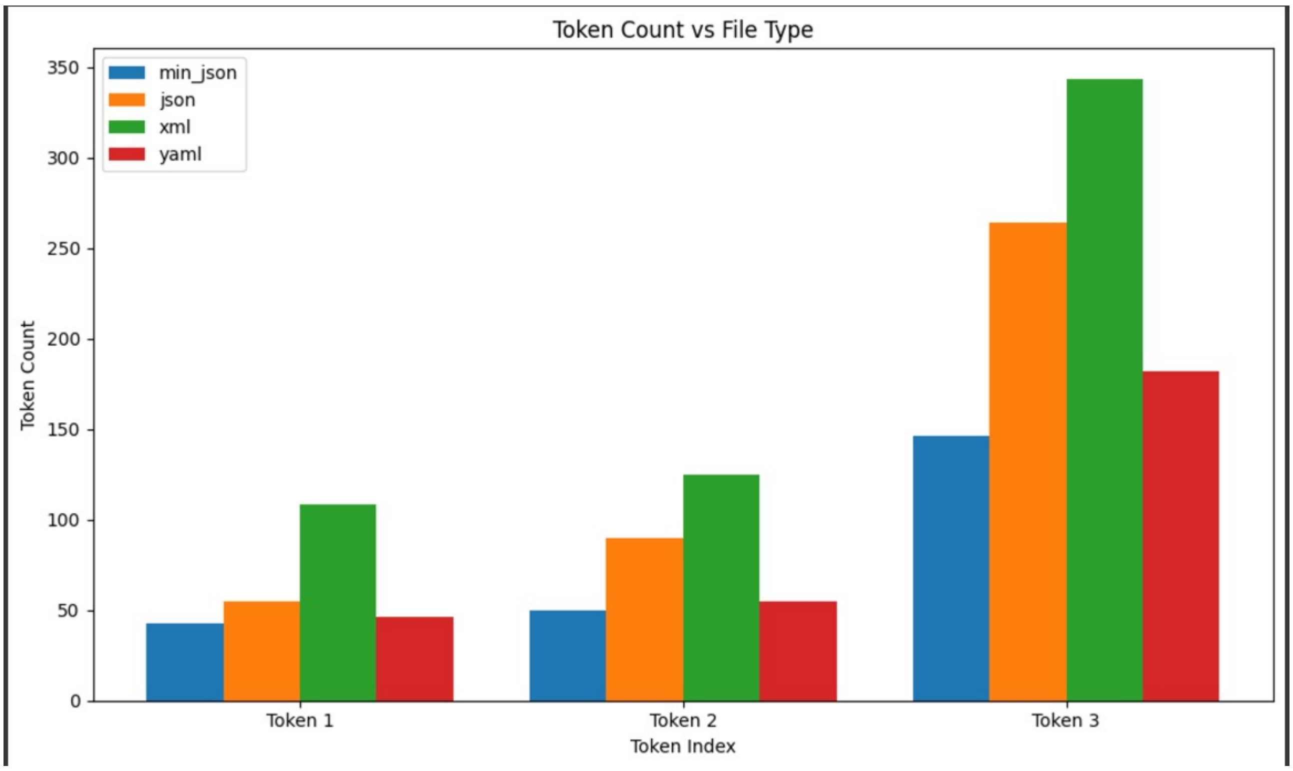


Figure 2:

2.2 Challenges

Below are the challenges faced after our mid-term progress.

1. Smaller LLMs for AI agents construct query parameters and request body inaccurately leading to incorrect retrieval and even failed requests in some cases.
2. The issue of latency persists but has been tackled to an extent by implementing caching.
3. The AI agent responsible for rephrasing the query and making it more semantically similar to the API documentation is a very crucial agent since it is at the very start of the pipeline and is responsible for enhancing retrieval. This huge dependency on one agent sometimes leads to inaccuracies in responses.

4. As mentioned in the "Future Works" section of our mid-term report, we had planned to make our RAG actionable - meaning it can take actions on behalf of the user. However, the major challenge we faced here was guardrails that ask the user for confirmation before actually taking an action.

2.3 Issues Addressed

1. **Agent Implementation:** DSPy enhances the generation of efficient agents by providing structured natural language signatures, which define each agent's task in clear, declarative terms, such as `question→answer` or `context, question→answer`.
2. **Speed-Accuracy Trade-off:** To address the issue of slow responses, we use smaller LLMs that are faster but occasionally inaccurate. This can be mitigated using fine-tuned LLMs in production environments.
3. **Latency:** Response time can be reduced by integrating a cache memory into each agent that stores their recent outputs. This allows us to answer repeated queries instantaneously, significantly reducing latency.
4. **Guardrails for Actions:** In addition to enabling our RAG to take actions on behalf of the user, we added guardrails to ensure that actions with potential high risk and consequences are executed only after an additional confirmation from the user.
5. **Scalability:** Current solutions that use embedding models to generate embeddings for every single row in a database face issues with keeping up with real-time data and high storage usage. Our approach generates embeddings only for the natural language documentation of API endpoints. This tackles the issues associated with storage usage.
6. **Passive Navigation:** Existing solutions primarily help users navigate to relevant information without synthesizing a direct answer, often requiring users to interpret the information themselves. By directly generating answers from API-retrieved data, our approach simplifies the user experience by providing answers rather than just navigation, making interactions more intuitive and reducing user effort.
7. **Security:** We retain the user's authentication details, such as JWT tokens or similar credentials, in the headers of each API call. This approach ensures that the user's access permissions are consistently enforced when retrieving data from the API.

3 User Interface

Our system features two carefully designed interfaces, each tailored to meet the needs of its specific audience. The user interface is simple and intuitive, catering to individuals with little or no technical background. It opens with a friendly widget and provides pop-ups to notify users when actions requiring permissions, such as placing an order, are initiated, ensuring transparency and user control. This interface goes beyond basic text responses by also supporting images and other visual elements, making interactions more engaging and informative. The developer interface is more advanced, offering detailed insights such as query rephrasing, accessed endpoints, retrieved data, latency, and token usage. These features enable developers to effectively monitor, evaluate, and improve the system. Together, these interfaces deliver an engaging user experience while equipping developers with robust tools for optimization.

4 Responsible AI Practices

To ensure our RAG meets the highest standards and gain user trust, we have implemented key practices to minimizes risks and create a responsible RAG system:

1) Irrelevant Data Filter: To safeguard user privacy and maintain relevance, we have implemented an Irrelevant Data Filter. This guardrail ensures that any sensitive/irrelevant JSON keys in the API response, which the client may not want to expose to users through the chatbot, are removed before the data is passed as context to the agents.

2) User approval for taking actions: This guardrails ensures that if the user query requires an action with potential long term consequences, explicit user consent is taken before executing the action

5 Key Insight Gained

1. **Importance of Guardrails and Safety Measures -** One of the most critical lessons that we learned while developing our Agentic RAG was the importance of integrating guardrails. We realized that the

system should operate between ethical and safe boundaries. Guardrails allowed us to filter out sensitive or irrelevant data, ensuring user privacy and data integrity. They also required user confirmation for actions that had significant, long-term consequences, such as financial or security-related decisions. This adds to the trust that the user builds while using our product. It reinforced the idea that, alongside performance, incorporating safety measures is essential for creating responsible, reliable AI systems

2. **Importance of Comprehensive Testing -** An essential takeaway is the critical role of comprehensive testing in deploying an effective solution. There is a wide choice of LLMs, embedding methods, indexing techniques,similarity scoring mechanisms etc. to choose from. Each combination of these components introduces unique trade-offs in terms of token usage, latency, accuracy, and cost-efficiency. For example, smaller LLMs may reduce costs and improve response times but can struggle with accuracy on more complex queries. Similarly, embedding strategies and indexing methods, such as vector-based search or hybrid retrieval, differ in their ability to manage large-scale data efficiently while maintaining low latency.

3. **Enabling Adaptive AI Systems with Real-Time Data Processing -**A key realization has been the importance of stream processing and real-time analytics in developing responsive and adaptive AI systems. In an environment where data changes rapidly, ensuring that systems can handle both streaming and batch data efficiently is vital for accurate, real-time decision-making. The ability to connect seamlessly with diverse data sources and process updates dynamically is what makes AI solutions truly effective.

   To manage these complexities, we have relied on Pathway, a powerful framework that combines ETL functionality, real-time indexing, and advanced AI pipeline support like LLMs and RAG. Its efficiency and flexibility have been instrumental in building robust, high-performance systems that meet the demands of live data processing.

   Thorough testing ensures that these elements are evaluated systematically across diverse use cases, enabling us to strike an optimal balance that meets both technical and operational goals.

6 Conclusion

Our Agentic RAG system has evolved into a specialized solution that transforms Retrieval-Augmented Generation by leveraging API endpoints for precise and secure data retrieval. By dynamically matching user queries with API documentation, the system ensures accurate endpoint selection and response generation while maintaining user-specific access controls. Through the integration of Pathway as a vector database, hybrid retrieval methods, and DSPy-based agent optimization, we've streamlined multi-agent workflows and reduced computational overhead. This approach not only addresses challenges like ambiguous queries and real-time data processing but also delivers direct, intelligent answers rather than basic navigation. Our focus on API-driven retrieval, security, and efficiency makes this system a robust and scalable solution for complex, data-rich environments

6.1 References

1. Pathway.com (n.d.). *Pathway.* GitHub. Retrieved from https://github.com/pathwaycom/pathway

2. Maleki, N., Padmanabhan, B., & Dutta, K. (2024). AI Hallucinations: A Misnomer Worth Clarifying. arXiv. https://doi.org/10.48550/arXiv.2401.06796

3. Ravuru, C., Sakhinana, S. S., & Runkana, V. (2024). Agentic Retrieval-Augmented Generation for Time Series Analysis. arXiv. https://doi.org/10.48550/arXiv.2408.14484

4. Wang, X., Wang, Z., Gao, X., Zhang, F., Wu, Y., Xu, Z., Shi, T., Wang, Z., Li, S., Qian, Q., Yin, R., Lv, C., Zheng, X., & Huang, X. (2024). Retrieval-augmented generation (RAG) techniques: Enhancing performance and efficiency through multimodal retrieval. arXiv. https://doi.org/10.48550/arXiv.2407.01219

5. Mehta, R., & Chilimbi, T. (2024). Amazon announces Rufus, a new generative AI-powered conversational shopping experience. Amazon. Retrieved from https://www.aboutamazon.com/news/retail/amazon-rufus.

6. Cook, A. (2024, September 02). *The Definitive Guide to Salesforce Einstein AI.* Salesforce. Retrieved from https://www.salesforce.com/news/press-releases/2023/03/07/einstein-generative-ai/

7. Wang, X., Wang, Z., Gao, X., Zhang, F., Wu, Y., Xu, Z., Shi, T., Wang, Z., Li, S., Qian, Q., Yin, R., Lv, C., Zheng, X., & Huang, X. (2024). Retrieval-augmented generation (RAG) techniques: Enhancing performance and efficiency through multimodal retrieval. arXiv. https://doi.org/10.48550/arXiv.2407.

01219 https://pathway.com/bootcamps/rag-and-llms/coursework/module-5-hands-on-development/docker-basics

8. Langchain Community, PathwayVectorClient. Langchain. https://python.langchain.com/api_reference/_modules/langchain_community/vectorstores/pathway.html#PathwayVectorClient
9. Unstructured.io, Unstructured: Making unstructured data accessible and usable. Unstructured.io. https://unstructured.io/
10. APIs.guru, APIs.guru: A collective list of public APIs. APIs.guru. https://apis.guru/
11. OpenBMB, Dragonball dataset. GitHub. https://github.com/OpenBMB/RAGEval/tree/main/dragonball_dataset
12. Pathway, Docker basics. Pathway. https://pathway.com/bootcamps/rag-and-llms/coursework/module-5-hands-on-development/docker-basics
13. Mostert, B., & Pelov, L. (2024, January 23). Announcing the OCI Generative AI Agents RAG service. Oracle. https://blogs.oracle.com/ai-and-datascience/post/oci-generative-ai-agents-rag-service