Domain-Specific Chatbot Powered by Retrieval-Augmented Generation (RAG)

Presented By: Nikolaos Lithoxopoulos

Motivation

Current Challenge

Traditional domain-specific RAG systems are designed for a fixed domain (e.g., medical, legal) and require significant manual effort to build and maintain knowledge bases for each domain. Our system allows users to **dynamically choose a domain** at the start, making it more versatile and adaptable across different contexts[1]

Solution

Developed a flexible, domain-specific RAG system that allows users to dynamically select a domain at the start of the interaction. The system then ensures that all interactions—whether questions or document uploads—remain strictly within that domain. Additionally, when searching the web for external information, the system restricts its retrieval to documents that are relevant and within the defined domain, ensuring consistently focused and accurate responses

Essential Criteria

- RAG system
- Specific Domain Focus
- Rephrase Questions
- Upload Sources (.pdf, .txt, .docx, & URLs)
- Show Metadata
- "I Don't Know" Responses
- Answer Arithmetic Reasoning Questions

LLMs
(Large Language Models)

LLMs are advanced AI models, that have been pre-trained on vast datasets to generate human-like text. LLMs are trained on vast datasets but are inherently limited by the **static nature of their training data,** cannot update their knowledge

Ask for up-to-date information -> **hallucination**

RAG
(Retrieval-Augmented Generation)

RAG enhances LLMs by incorporating an additional retrieval step. Before generating a response, the system retrieves relevant documents from a knowledge base (or external sources) to ground the response in factual information.

RAG -> reducing **hallucinations** and increasing **accuracy**.[2]

Frameworks

Streamlit

Provides a user-friendly interface to interact with the system, making it easy for users to ask questions and visualize responses.

Ollama

Runs large language models (LLMs) locally for efficient and private processing.

Meta

Llama3.1 8B

LangChain

LangChain is a powerful platform designed to use the capabilities of large language models for various applications.

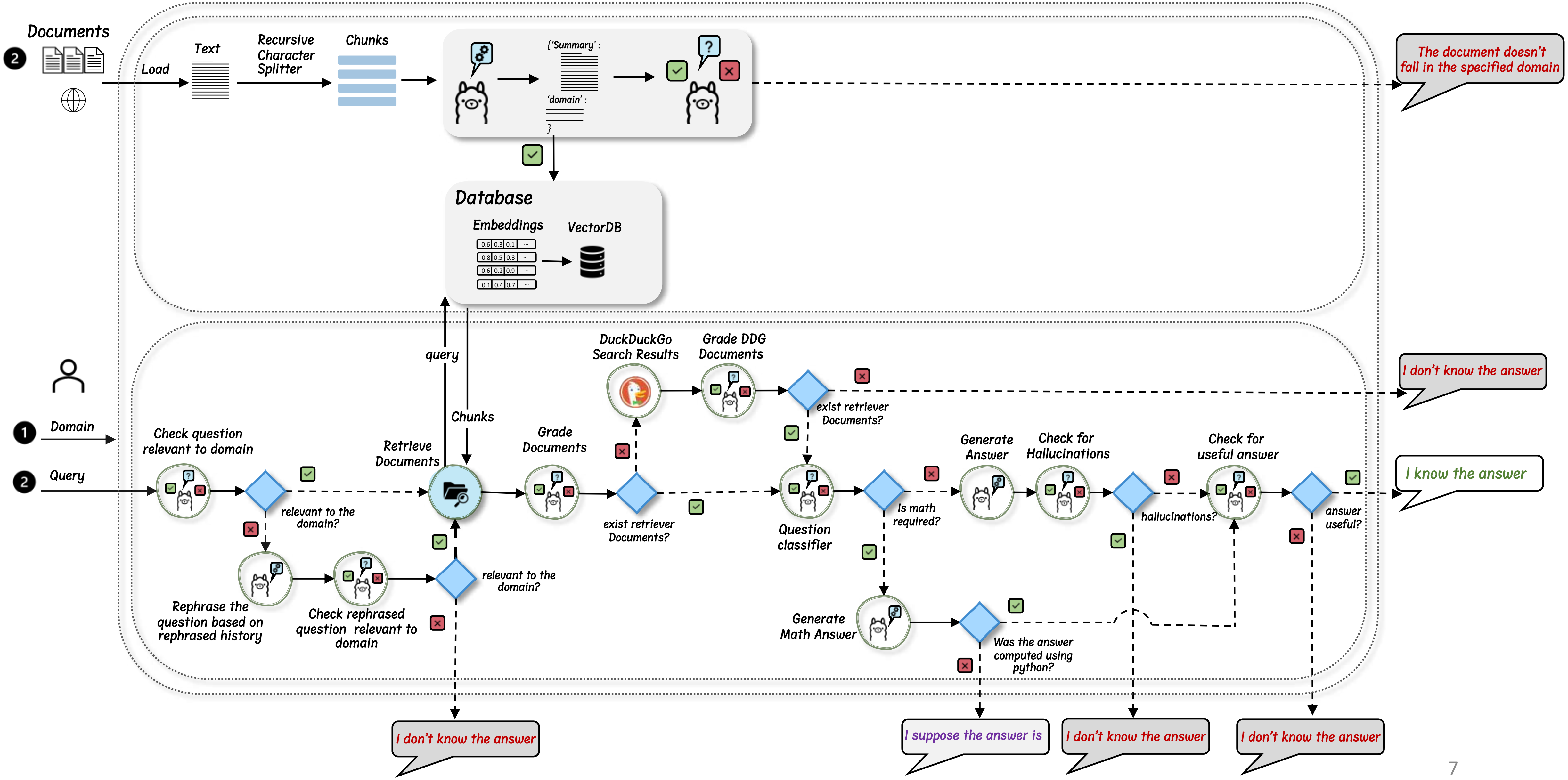
- Integration with LLMs,
- Modular Architecture,
- Wide Range of Applications,
- Developer-Friendly,
- Scalability
- Community and Support

Milvus DB

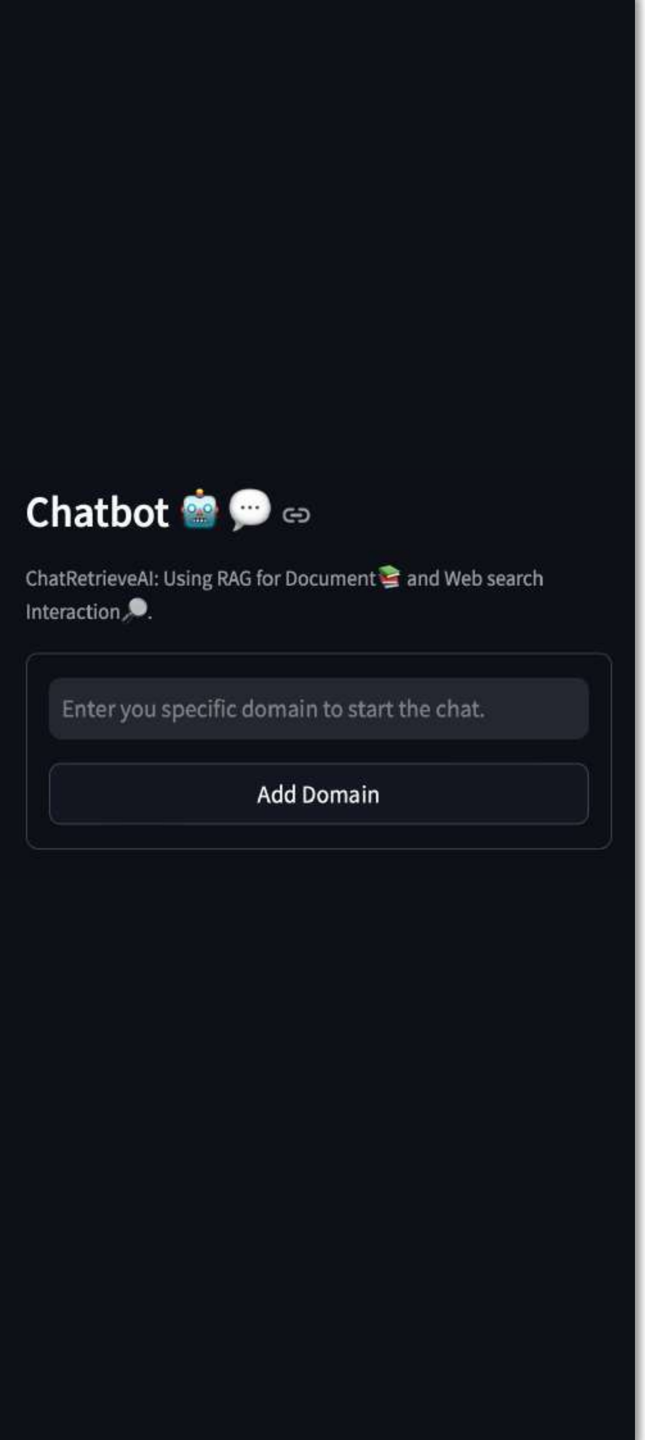
Stores and manages high-dimensional vectors, enabling fast and accurate document retrieval

System Architecture
&
Implementation Overview

Overview



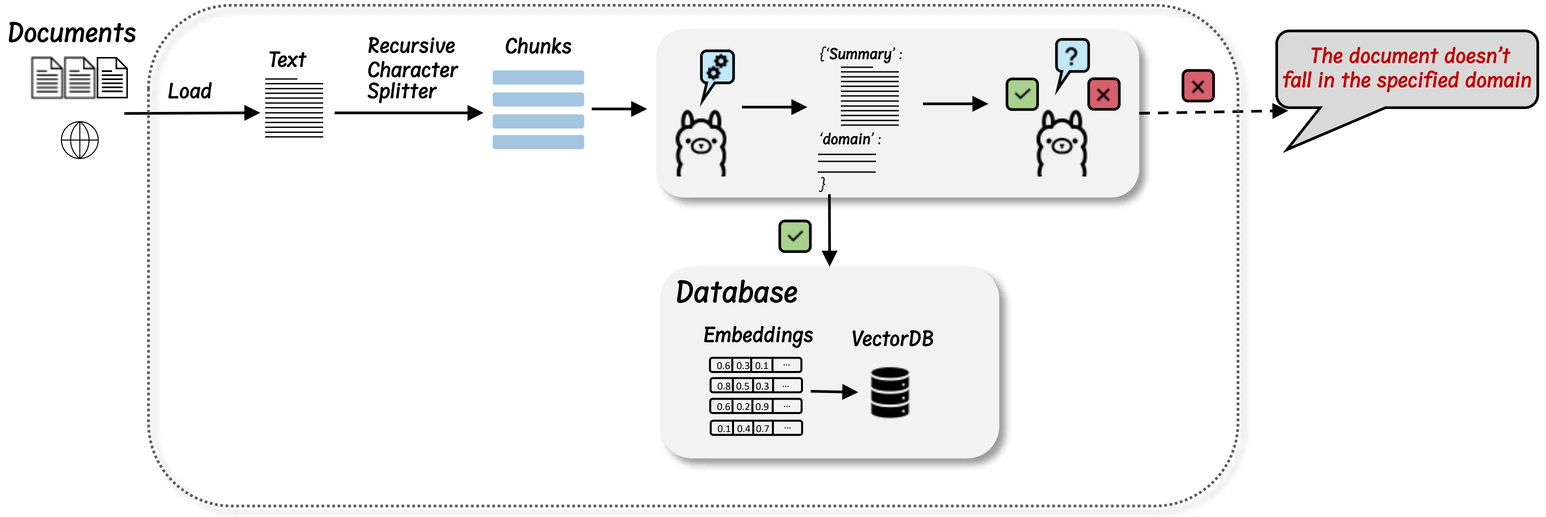
Documents
2
Load
Text
Recursive Character Splitter
Chunks
{'Summary' :
'domain' :
}
The document doesn't fall in the specified domain
Database
Embeddings
VectorDB
0.6 0.3 0.1 ...
0.8 0.5 0.3 ...
0.6 0.2 0.9 ...
0.1 0.4 0.7 ...
1
Domain
2
Query
query
Chunks
Check question relevant to domain
relevant to the domain?
Retrieve Documents
Grade Documents
exist retriever Documents?
DuckDuckGo Search Results
Grade DDG Documents
exist retriever Documents?
I don't know the answer
Question classifier
Is math required?
Generate Answer
Check for Hallucinations
hallucinations?
Check for useful answer
answer useful?
I know the answer
Rephrase the question based on rephrased history
Check rephrased question relevant to domain
relevant to the domain?
Generate Math Answer
Was the answer computed using python?
I don't know the answer
I suppose the answer is
I don't know the answer
I don't know the answer



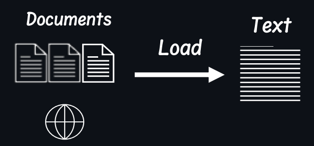
Domain Customization

Users can specify a domain of interest at the beginning of the session, ensuring that the chatbot focuses on relevant content and answers within the selected domain.

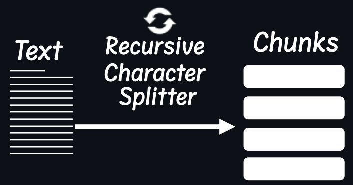
Document processing



Load text



Text to Chunks



Garbage In, Garbage Out

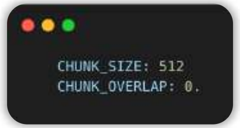
The "Garbage In, Garbage Out" principle applies here: properly cleaning the text is essential to minimize errors and improve response accuracy.

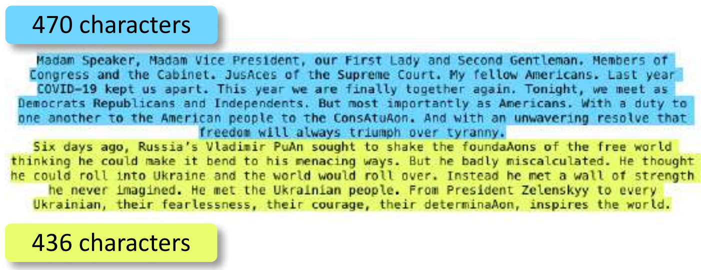
Applying a custom cleaning function to ensure the text is in the optimal format for the chatbot's processing. This function performs the following tasks:

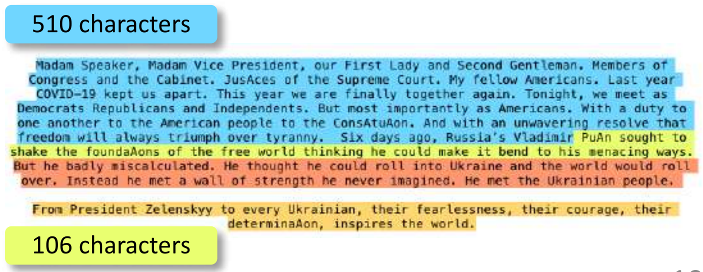
- Removes Unnecessary Newlines
- Eliminates Excessive Whitespace:
- Adjusts existing metadata to fit the required project format

Recursive Character Text Splitter overview

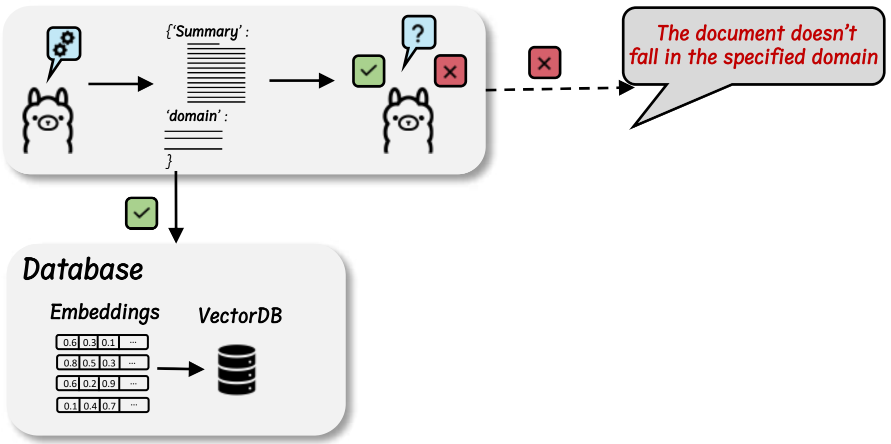
- Advanced technique using text structure, preserves natural text structure
- Recursively splits text with separators (paragraphs, sentences, etc.).
- Maintains logical boundaries in chunks
- Chunk size: 512 [3]
- Overlap: 51 [3]







Documents Domain Detection and Validation



The system processes the document chunks and uses a model chain to return a summary and the list of 3 possible domains. These detected domains are then compared to the user-defined source domain.

- If the domains do not match, the document is rejected.
- If the domains match, the documents are added to the vector database.

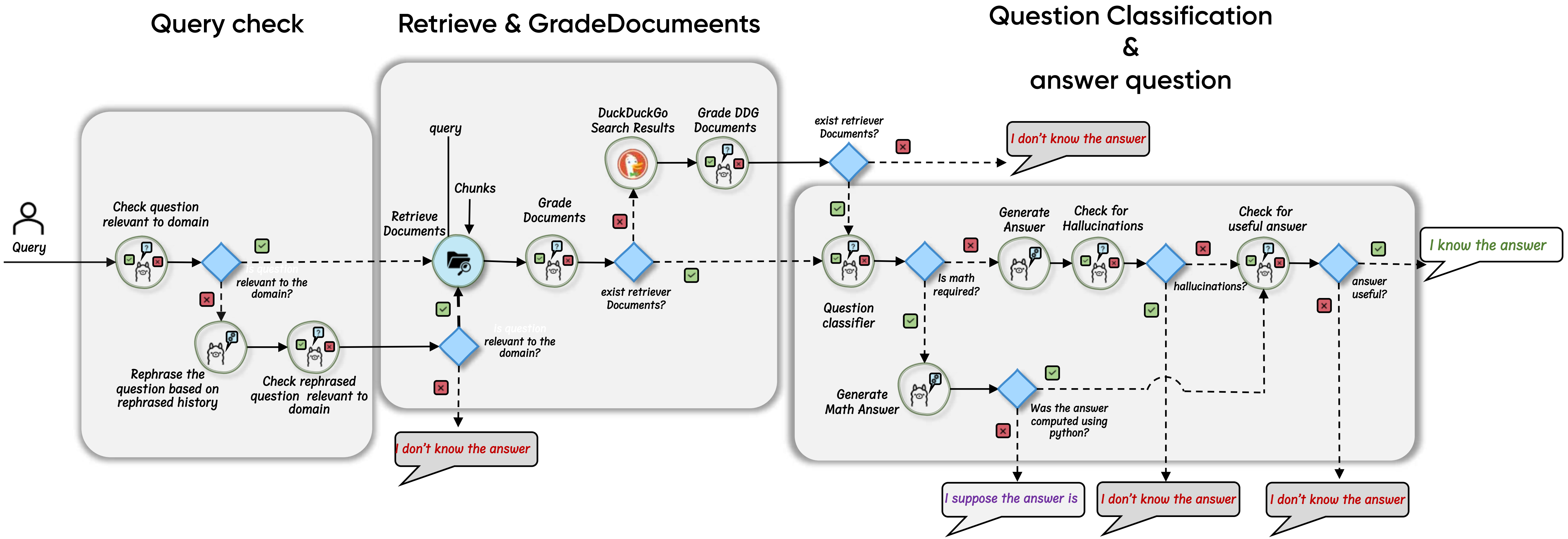
This method ensures that only documents relevant to the specific domain are incorporated into the system, maintaining data quality and relevance.

Milvus DB

The vector database is created using the HuggingFaceBgeEmbeddings() method for embeddings, and a retriever is created from the vector store using the default similarity search type and the default number of documents to retrieve ('k':4).

```
class VectorDB:
    def __init__(self):
        print("vectordb.py - __init__()")

        model_name = "BAAI/bge-large-en"
        model_kwargs = {'device': 'cpu'}
        encode_kwargs = {'normalize_embeddings': True}
        self.hf = HuggingFaceBgeEmbeddings(
            model_name=model_name,
            model_kwargs=model_kwargs,
            encode_kwargs=encode_kwargs
        )

        self.vector_store =  Milvus(
            collection_name = cfg.COLLECTION_NAME,
            embedding_function= self.hf,
            connection_args={"uri": cfg.URI},
            drop_old = True
        )

        self.retriever = self.vector_store.as_retriever()


    def add_documents(self, chunks: List[Document]):
        print("vectordb.py - add_documents()")

        uuids = [str(uuid4()) for _ in range(len(chunks))]
        self.vector_store.add_documents(documents=chunks, ids=uuids)rn go(f, seed, [])
}
```

Query Processing



LangGraph 🦜🕸️

For the query proccesing, Langgraph from Langchain is being used. [4]

Reliability: Predefined control flows, more reliable than dynamic LLM-based agents

Custom State: Maintain custom state across nodes to control behaviour

Debugging: Easier to debug with clear, defined flow

Flow: Well-defined flow, easier to describe and implement

```python
class GraphState(TypedDict):
    """
    Represents the state of our graph.

    """
    print('\n--- GRAPH STATE ---')
    question: str
    rephrase_question: str
    q_domain_relevance: str
    documents: List[str]
    grade_documents: List[str]
    domain : str
    hallucination: str
    generation_score: str
    question_type: str
    execution_path: List[str] = []
    answer_useful: str
    answer: str
```

```python
class WorkflowInitializer:

    def __init__(self, system):
        self.system = system

    def initialize(self):
        print('\nCalling => langgraph.py - WorkflowInitializer.initialize()')
        workflow = StateGraph(self.system.GraphState)

        workflow.set_entry_point("check_query_domain")
        workflow.add_node("check_query_domain", self.system._check_query_domain)
        workflow.add_node("rephrase_based_history", self.system._rephrase_query)
        workflow.add_node("retrieve", self.system._retrieve)
        workflow.add_node("grade_docs", self.system._grade_documents)
        workflow.add_node("grade_ddg_docs", self.system._grade_documents)
        workflow.add_node("check_query_domain_end", self.system._check_query_domain)
        workflow.add_node("generate", self.system._generate)
        workflow.add_node("ddg_search", self.system._ddg_search)
        workflow.add_node("answer_check", self.system._answer_check)
        workflow.add_node("hallucination_check", self.system._hallucination_check)
        workflow.add_node("question_classification", self.system._question_classifier)
        workflow.add_node("math_generate", self.system._math_generate)

        workflow.add_conditional_edges(
            "check_query_domain",
            lambda state: state["q_domain_relevance"],
            {
                "yes": "retrieve",
                "no": "rephrase_based_history",
            },
        )

        workflow.add_edge("rephrase_based_history", "check_query_domain_end")
```

Propmts

Llama3.1

Our system support all available models from Ollama, and we are currently utilizing **LLama 3.1**. Following META's best practices, we have updated the prompts to the new format, ensuring to obtain the best results with **LLama 3.1** by incorporating the recommended specific tokens. [5]

```
query_domain_check =PromptTemplate(
    template="""<|begin_of_text|><|start_header_id|>system<|end_header_id|>
    You are a grader assessing whether a user question falls within the specified {domain} domain.
    Your task is to determine if the question is directly related to {domain} by considering the content
    and context of the question.
    Give a binary score 'yes' or 'no' score to indicate whether the domain is relevant to the question.
    Provide the binary score as a JSON with a single key 'score' and no premable or explanation.<|eot_id|>
    <|start_header_id|>user<|end_header_id|>
    Here is the user question: {question} <|eot_id|>
    <|start_header_id|>assistant<|end_header_id|>
    """,
    input_variables=["question", "domain"],
)
```

```
<|begin_of_text|>
  <|end_of_text|>
```

Specifies the start of the prompt
Model will cease to generate more tokens.
This token is generated only by the base models.

```
<|start_header_id|>
  <|end_header_id|>
```

These tokens enclose the role for a particular message.
The possible roles are: [**system, user, assistant and ipython**]

```
<|eot_id|>
```

Represents when the model has determined that it has finished interacting with the user message that initiated its response.

In our Chatbot we use Zero-Shot and Chain-of-Thought (CoT) prompting techniques

Always with structure output.

```
response_schemas_domain_check = [
    ResponseSchema(
        name="score",
        description="Answer with 'yes' if the question is related to the domain, 'no' otherwise."),
]


class AnswerWithSources(BaseModel):
    """An answer to the question, with keys 'answer' and 'sources'.
        'answer' : The answer to user's question.
        'sources' : The key 'sources' of metadata of the Documents that are used to generate the answer."""
    answer: str
    sources: Annotated[
        Set[str],
        ...,
        "The key 'sources' of the Metadata",
    ]
```

Question check relevant to domain

LLM calls to diffine if the question is relevant to the domain using the appropriate instructions prompt.

Zero-Shot Prompting

```
"""
You are a grader assessing whether a user question
falls within the specified {domain} domain.
User question: {question}
"""
```

Rephrase the question based on rephrased history

The chatbot rephrases questions to standalone versions based on chat history. Two chat histories are maintained:

- **chat_history**
- **chat_rephrased_history**

This approach ensures better results and clearer responses.

chat_history

- Is the camera service already implemented?
- What the use of it ?

  What is the purpose of the Camera Service?
- Is the Face Recognition already implemented?
- What the use of it ?

  What is the purpose of the Face Recognition and Camera Service?

chat_rephrased_history

- Is the camera service already implemented?
- What the use of it ?

  What is the purpose of the Camera Service?
- Is the Face Recognition already implemented?
- What the use of it ?

  What is the purpose of the Face Recognition?

`create_history_aware_retriever`

Purpose

Enhances retrieval by integrating conversation history
for more contextually relevant document results.

Key Concepts:

- **History-Aware:** Incorporates chat history to refine search queries.
- **LLM-Assisted Retrieval**: Uses a language model (LLM) to generate search queries from chat history.
- **Dynamic Interaction:** If no chat history is present, the retriever directly processes the input; otherwise, LLM generates a search query based on the conversation.
- **Without Chat History:** Input directly passed to the retriever for document search.

Grading Documents

- Utilizing LLM for Document Grading
- Filters out irrelevant documents before passing them to the model for answer generation.
- Ensures that only the most relevant information is used, enhancing system efficiency.
- **Decision-Making:** Triggers a web search if the number of relevant documents is insufficient.

Web-search

If **no** documents are available, the system performs a web search.

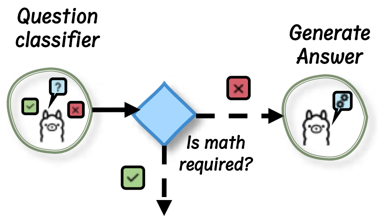
Use DuckDuckGoSearchResults for more additional information (e.g title and link)

```
"[snippet: Machine learning in healthcare helps medical …,
"title: 23 Machine Learning in,
"link:
'https://builtin.com/artif...]"
```

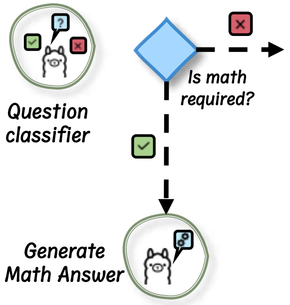
Question answering

The question answering depends on whether the question classification score is ”text" or ”math"

If "text": The grade_documents and user query are sent directly to the LLM to generate an answer.



If "math": The grade_documents and user query are processed by the LLM, along with a step-wise reasoning prompt, which breaks down the solution step-by-step.



Expression Generation: The system aims to return a NumExpr-compatible expression that can be efficiently resolved using ne.evaluate().

Hallucinations check

A hallucination check node is integrated as an additional security measure. While the RAG (Retrieval-Augmented Generation) approach is already designed to minimize hallucinations by grounding responses in retrieved data, this additional step further enhances the system's reliability and accuracy.

Usefull answer

An additional node is implemented to evaluate whether the generated answer is useful. This check operates independently from the hallucination detection, ensuring that even if an answer is factually correct, it is also relevant and helpful to the user's query. This dual-layer validation enhances both the accuracy and the practical value of the system's responses

Arithmetic Reasoning Question answering

For **arithmetic reasoning**, we implement the **Chain-of-Thought (CoT)** prompting technique [6], guiding the LLM to generate an expression that can be evaluated using the numexpr library's **np.evaluate()** function.

If the system is unable to resolve the expression, it will retrieve the solution from an online source.

To maintain transparency, the user is informed whether the solution was computed locally using Python or retrieved externally from the web.

```
math_solver = PromptTemplate(
    template="""
    <|begin_of_text|><|start_header_id|>system<|end_header_id|>
    You are an expert AI specialized in generating expressions for ne.evaluate(.) to solve arithmetic tasks.
    Given specific numbers, write a NumExpr-compatible expression that directly calculates the result.
    The final expression must contain only numbers and operators, with no variables.

    Related Documents: {documents}\n

    Example Task:
    Related Documents: [Cappuccinos cost $2, iced teas cost $3, cafe lattes cost $1.5 and espressos cost $1 each.]
    Task: Sandy orders some drinks for herself and some friends. She orders three cappuccinos,
    two iced teas, two cafe lattes, and two espressos. How much change does she receive back for a twenty-dollar bill?
    Answer: {{
    'step-wise reasoning': [
        'Calculate the total cost of three cappuccinos: 3 * 2',
        'Calculate the total cost of two iced teas: 2 * 3',
        'Calculate the total cost of two cafe lattes: 2 * 1.5',
        'Calculate the total cost of two espressos: 2 * 1',
        'Sum these costs to get the total expenditure',
        'Subtract the total expenditure from 20 to find the change'
    ],
    'expr': '20 - (3 * 2 + 2 * 3 + 2 * 1.5 + 2 * 1)'
    }}
    <|eot_id|><|begin_of_text|><|start_header_id|>user<|end_header_id|>
    Task: {question}
    <|eot_id|><|start_header_id|>assistant<|end_header_id|>
     """,
    input_variables=["question", "documents"],
)
```

```
answer_to_neEvaluate = ne.evaluate(expr_str)
state['answer'] = {
  "answer": f"{stepwise_str}\n\n Final answer: {answer_to_neEvaluate} ",
  "metadata": sources,
  "calculation": 'Computed with python'}

#=================================================================

state['answer'] = {
  "answer": f"{stepwise_str}\n\n Final answer: {solution} ",
  "metadata": sources,
  "calculation": 'Not python computed or Web based solution, \n
                  "maybe not be accurate. (Check the sources)🚨 '}
```

DEMO

Testing

Unitest

For the testing process, two main categories of unit tests have been implemented for the chatbot

Edges test

```python
def test_positive_domain_relavance(self):
    question = "How can AI improve basketball training?"
    inputs = {"question": question, "domain": self.domain}
    expected_classification = 'yes'

    state = self.kbs._check_query_domain(inputs)

    print(f"\n{self.__class__.__name__}:        {inspect.currentframe().f_code.co_name}")
    print(f"\nCheck_query_domain -> State:      {state}")

    self.assertEqual(state['question'], question)
    self.assertEqual(state['q_domain_relevance'], expected_classification)


def test_negative__domain_relavance(self):
    question = "How is the weather today?"
    inputs = {"question": question, "domain": self.domain}
    expected_classification = 'no'

    state = self.kbs._check_query_domain(inputs)

    print(f"\n{self.__class__.__name__}:        {inspect.currentframe().f_code.co_name}")
    print(f"\nCheck_query_domain -> State:      {state}")

    self.assertEqual(state['question'], question)
    self.assertEqual(state['q_domain_relevance'], expected_classification)
```

Path Execution Test

```python
def test_path_1(self):
        question = "What are the benefits of using wearable devices
                    'in basketball training and performance analysis?"
        expected_execution_path = [
            'check_query_domain',
            'retrieve',
            'grade_docs',
            'question_classification',
            'generate',
            'hallucination_check',
            'answer_check',
        ]

        inputs = {"question": question, "domain": self.domain}
        state = self.chat_pdf.invoke(inputs)

        # Check if the question is in the spacified domain
        self.assertEqual(state['q_domain_relevance'], 'yes', msg="The question is not in the specified domain")

        # Check if the exist positive grade documents
        self.assertTrue(len(state['grade_documents']) > 0, msg="No positive grade documents")

        # Check if the question is correctly classified as non-math
        self.assertEqual(state['question_type'], 'no', msg="The question is classified as math")

        # Check if the answer is generated and does not contain hallucinations
        self.assertEqual(state['hallucination'], 'no', msg="The answer contains hallucinations")

        # Check if the answer is not useful
        self.assertEqual(state['answer_useful'], 'useful', msg="The answer is not useful")

        # Check if the execution path is as expected until the last expected step
        self.assertEqual(state['execution_path'], expected_execution_path)
```

Thank You

References

[1] S. Wang, J. Liu, S. Song, J. Cheng, Y. Fu, P. Guo, K. Fang, Y. Zhu, and Z. Dou, "DomainRAG: A Chinese Benchmark for Evaluating Domain-specific Retrieval-Augmented Generation," *arXiv preprint arXiv:2406.05654v2*, 2024.

[2] P. Béchard and O. M. Ayala, "Reducing hallucination in structured outputs via Retrieval-Augmented Generation," arXiv preprint arXiv:2404.08189v1, 2024.

[3] Zilliz, "Exploring RAG, Chunking, LLMs, and Evaluations," Zilliz Blog, 2024. [Online]. Available: https://zilliz.com/blog/exploring-rag-chunking-llms-and-evaluations. [Accessed: Aug. 26, 2024].

[4] C. Jeong, "A Study on the Implementation Method of an Agent-Based Advanced RAG System Using Graph," Journal of Information Technology Services, vol. 19, no. 4, pp. 31-47, 2023.

[5] Meta, "LLaMA 3 Model Cards and Prompt Formats," Meta AI Documentation, 2024. [Online]. Available: https://llama.meta.com/docs/model-cards-and-prompt-formats/llama3_1/.

[6] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. H. Chi, Q. V. Le, and D. Zhou, "Chain-of-Thought Prompting Elicits Reasoning in Large Language Models," arXiv preprint arXiv:2201.11903v6, 2023.