**ResearchPal AI: An Extensible Framework for Advanced Retrieval-Augmented Generation**

**Introduction**

ResearchPal AI is an innovative framework designed to address the limitations of current Large Language Models (LLMs) and Retrieval-Augmented Generation (RAG) systems. It offers a flexible, extensible architecture that allows for customization at various levels, from knowledge base management to query processing and evaluation.

**Key Features and System Architecture**

**1. Dynamic Knowledge Base**

ResearchPal AI employs a dynamic knowledge base that can be continuously updated and expanded:

* **PDF Ingestion**: Allows uploading of domain-specific documents.
* **Wikipedia Integration**: Dynamically fetches and integrates up-to-date information.
* **Manual Input**: Supports direct input of specialized knowledge.

python

Copy

class SelfRAGSystem:

def ingest\_pdf(self, pdf\_path: str) -> int:

loader = PyPDFLoader(pdf\_path)

documents = loader.load()

text\_splitter = RecursiveCharacterTextSplitter(chunk\_size=1000, chunk\_overlap=200)

texts = text\_splitter.split\_documents(documents)

added\_docs = self.knowledge\_base.add\_documents(texts)

return len(added\_docs)

**2. Multi-Stage Query Processing**

The system implements a sophisticated query processing pipeline:

* **Query Classification**: Categorizes queries to determine the most appropriate retrieval strategy.
* **Contextual Retrieval**: Uses FAISS for efficient, similarity-based information retrieval.
* **Answer Generation**: Leverages advanced language models for response generation.
* **Answer Evaluation**: Assesses the quality of generated responses.

python

Copy

class AnswerEvaluator:

def evaluate(self, query: str, answer: str) -> AnswerEvaluation:

eval\_prompt = PromptTemplate.from\_template(

"Evaluate the following answer for relevance, completeness, and accuracy:\n\n"

"Query: {query}\nAnswer: {answer}\n\n"

"Provide scores (0-1) and explanations for each criterion."

)

eval\_result = self.llm(eval\_prompt.format(query=query, answer=answer))

*# Parse eval\_result and return AnswerEvaluation object*

**3. Iterative Query Refinement**

To improve performance on complex queries, the system can refine queries based on initial results:

python

Copy

class QueryEnhancer:

def enhance\_query(self, original\_query: str, context: str, previous\_answer: str, evaluation: AnswerEvaluation) -> str:

enhance\_prompt = PromptTemplate.from\_template(

"Given the original query, context, previous answer, and evaluation, suggest an improved query:\n\n"

"Original Query: {original\_query}\nContext: {context}\n"

"Previous Answer: {previous\_answer}\nEvaluation: {evaluation}\n\n"

"Improved Query:"

)

return self.llm(enhance\_prompt.format(

original\_query=original\_query,

context=context,

previous\_answer=previous\_answer,

evaluation=evaluation

))

**4. Transparent Processing**

The system offers a debug mode that provides insights into the decision-making process:

python

Copy

def process\_query(self, query, message\_placeholder):

self.log\_action("Query Received", f"Query: {query}")

try:

response, iterations = self.rag\_system.query(query)

full\_response = f"\*\*Answer:\*\* {response}\n\n"

if st.session\_state.debug\_mode:

full\_response += "\*\*Process Details:\*\*\n"

for iteration in iterations:

full\_response += f"Iteration {iteration.get('iteration', 'N/A')}:\n"

full\_response += f"- Strategy: {iteration.get('strategy', 'N/A')}\n"

full\_response += f"- Explanation: {iteration.get('explanation', 'N/A')}\n"

*# ... more debug information ...*

return full\_response

except Exception as e:

error\_message = f"Error processing query: {str(e)}"

self.log\_action("Query Processing Failed", error\_message)

return error\_message

**Extensibility and Customization**

ResearchPal AI is designed as a flexible framework that can be extended and customized to meet specific needs. Here's how you can leverage its extensibility:

**1. Federated Knowledge Structure**

Instead of centralizing all information, ResearchPal AI supports a federated approach:

python

Copy

class FederatedKnowledgeBase:

def \_\_init\_\_(self):

self.local\_store = FAISS.from\_texts(["Initial empty knowledge base"], self.embeddings)

self.external\_sources = {}

def add\_external\_source(self, name: str, source: Callable):

self.external\_sources[name] = source

def query(self, query: str, sources: List[str] = ["local"]):

results = []

if "local" in sources:

results.extend(self.local\_store.similarity\_search(query))

for source in sources:

if source in self.external\_sources:

results.extend(self.external\_sources[source](query))

return results

*# Usage*

knowledge\_base = FederatedKnowledgeBase()

knowledge\_base.add\_external\_source("enterprise\_db", query\_enterprise\_database)

knowledge\_base.add\_external\_source("pubmed", query\_pubmed\_api)

This structure allows easy integration with enterprise knowledge bases or public databases without copying all data locally.

**2. Custom Tool Integration**

You can extend the system's capabilities by adding custom tools:

python

Copy

class SelfRAGSystem:

def add\_tool(self, tool: Tool):

self.tools.append(tool)

if self.agent\_executor:

self.agent\_executor.tools.append(tool)

*# Example: Adding a custom PubMed search tool*

class PubMedSearchTool:

def search\_pubmed(self, query: str) -> str:

*# Implement PubMed search logic here*

pass

def get\_tool(self) -> Tool:

return Tool(

name="PubMed Search",

func=self.search\_pubmed,

description="Search PubMed for medical research papers"

)

rag\_system = SelfRAGSystem()

pubmed\_tool = PubMedSearchTool()

rag\_system.add\_tool(pubmed\_tool.get\_tool())

**3. Pluggable Evaluation and Classification**

The evaluation and classification mechanisms can be customized:

python

Copy

class SelfRAGSystem:

def set\_answer\_evaluator(self, evaluator: BaseEvaluator):

self.answer\_evaluator = evaluator

def set\_query\_classifier(self, classifier: BaseClassifier):

self.query\_classifier = classifier

*# Custom Evaluation Example*

class SentimentBasedEvaluator(BaseEvaluator):

def evaluate(self, query: str, answer: str) -> AnswerEvaluation:

sentiment = analyze\_sentiment(answer)

return AnswerEvaluation(

relevance\_score=sentiment.relevance,

completeness\_score=sentiment.completeness,

accuracy\_score=sentiment.accuracy

)

*# Custom Classification Example*

class DomainSpecificClassifier(BaseClassifier):

def classify(self, query: str) -> str:

if "medical" in query.lower():

return "medical\_rag"

elif "legal" in query.lower():

return "legal\_rag"

else:

return "general\_rag"

rag\_system.set\_answer\_evaluator(SentimentBasedEvaluator())

rag\_system.set\_query\_classifier(DomainSpecificClassifier())

**4. Dynamic Query Enhancement**

When the local knowledge base is insufficient, the system can dynamically query external sources:

python

Copy

class QueryProcessor:

def process\_query(self, query: str) -> str:

local\_answer = self.query\_local\_knowledge\_base(query)

if self.answer\_evaluator.is\_satisfactory(local\_answer):

return local\_answer

enhanced\_query = self.query\_enhancer.enhance(query, local\_answer)

external\_answer = self.query\_external\_sources(enhanced\_query)

return self.combine\_answers(local\_answer, external\_answer)

def query\_external\_sources(self, query: str) -> str:

for tool in self.external\_tools:

if tool.is\_relevant(query):

return tool.execute(query)

return ""

**Practical Implementation**

Here's how you might use these features in practice:

python

Copy

*# Initialize the system*

rag\_system = SelfRAGSystem()

*# Add custom knowledge sources*

rag\_system.add\_external\_source("enterprise\_db", EnterpriseDBConnector())

rag\_system.add\_external\_source("pubmed", PubMedAPIConnector())

*# Add custom tools*

rag\_system.add\_tool(CustomCalculatorTool().get\_tool())

rag\_system.add\_tool(DomainSpecificSearchTool().get\_tool())

*# Set custom evaluation and classification*

rag\_system.set\_answer\_evaluator(IndustrySpecificEvaluator())

rag\_system.set\_query\_classifier(MultiLabelClassifier())

*# Use the system*

query = "What are the latest treatments for type 2 diabetes?"

answer, process\_details = rag\_system.query(query)

print(f"Answer: {answer}")

print("Process Details:")

for step in process\_details:

print(f"- {step['description']}: {step['result']}")

**Conclusion**

ResearchPal AI offers a comprehensive solution to the limitations of current LLM and RAG systems. Its flexible architecture allows for customization at every level, from knowledge base management to query processing and evaluation. By providing a federated knowledge structure, an extensible tool ecosystem, and pluggable components, it enables the creation of specialized, adaptive AI research assistants tailored to specific domains and use cases.

Whether you're integrating with enterprise systems, adding domain-specific tools, or implementing custom evaluation criteria, ResearchPal AI provides the flexibility to build a system that meets your unique requirements while addressing the common challenges in information retrieval and knowledge synthesis.