following above discussion, please give example python codes for each type of RAG

following above discussion, please give example python codes for each type of RAG

Here are example Python codes for each type of RAG architecture:

1. Simple RAG

```
from langchain import hub
from langchain_core.documents import Document
from langchain_core.vectorstores import InMemoryVectorStore
from langchain_text_splitters import RecursiveCharacterTextSplitter
from langgraph.graph import START, StateGraph
from typing_extensions import Annotated, List, TypedDict

# Load and chunk contents of the blog
loader = WebBaseLoader(
    web_paths=("https://lilianweng.github.io/posts/2023-06-23-agent/",),
    bs_kwargs=dict(
        parse_only=bs4.SoupStrainer(
            class_=("post-content", "post-title", "post-header")
        )
    ),
)
docs = loader.load()
text_splitter = RecursiveCharacterTextSplitter(chunk_size=1000, chunk_overlap=200)
all_splits = text_splitter.split_documents(docs)

# Index chunks
vector_store = InMemoryVectorStore(embeddings)
_ = vector_store.add_documents(all_splits)

# Define schema for search
class Search(TypedDict):
    query: Annotated[str, ..., "Search query to run."]
    section: Annotated[
        Literal["beginning", "middle", "end"],
        ...,
        "Section to query.",
    ]

# Define prompt for question-answering
prompt = hub.pull("rlm/rag-prompt")
```

```
# Define state for application
class State(TypedDict):
    question: str
    context: List[Document]
    answer: str

# Define application steps
def retrieve(state: State):
    retrieved_docs = vector_store.similarity_search(state["question"])
    return {"context": retrieved_docs}

def generate(state: State):
    docs_content = "\n\n".join(doc.page_content for doc in state["context"])
    messages = prompt.invoke({"question": state["question"], "context": docs_content})
    response = llm.invoke(messages)
    return {"answer": response.content}

# Compile the graph
graph_builder = StateGraph(State).add_sequence([retrieve, generate])
graph_builder.add_edge(START, "retrieve")
graph = graph_builder.compile()

# Run the graph
result = graph.invoke({"question": "What is the financial outlook for ZEKO?"})
print(result["answer"])
```

2. Simple RAG with Memory

```
# This example extends the Simple RAG by adding a memory component to store previous inte

# Define a memory class to store previous interactions
class Memory:
    def __init__(self):
        self.history = []

    def add(self, interaction):
        self.history.append(interaction)

    def get(self):
        return self.history

# Modify the State to include memory
class State(TypedDict):
    question: str
    context: List[Document]
    answer: str
    memory: Memory

# Update the retrieve function to use memory
def retrieve(state: State):
    retrieved_docs = vector_store.similarity_search(state["question"])
    state["memory"].add({"question": state["question"], "context": retrieved_docs})
    return {"context": retrieved_docs}

# Update the generate function to use memory
```

```
def generate(state: State):
    docs_content = "\n\n".join(doc.page_content for doc in state["context"])
    messages = prompt.invoke({"question": state["question"], "context": docs_content})
    response = llm.invoke(messages)
    state["memory"].add({"answer": response.content})
    return {"answer": response.content}

# Initialize memory
memory = Memory()

# Compile the graph with memory
graph_builder = StateGraph(State).add_sequence([retrieve, generate])
graph_builder.add_edge(START, "retrieve")
graph = graph_builder.compile()

# Run the graph with memory
result = graph.invoke({"question": "What is the financial outlook for ZEKO?", "memory": m
print(result["answer"])
```

3. Branching RAG

```
# This example demonstrates branching RAG where the system decides which data source to

# Define different data sources
class DataSource:
    def __init__(self, name):
        self.name = name

    def query(self, query):
        # Placeholder for querying different data sources
        return f"Querying {self.name} for {query}"

# Define a function to decide which data source to use
def decide_data_source(query):
    if "financial" in query.lower():
        return DataSource("Financial Database")
    elif "medical" in query.lower():
        return DataSource("Medical Database")
    else:
        return DataSource("General Database")

# Modify the retrieve function to use branching
def retrieve(state: State):
    data_source = decide_data_source(state["question"])
    retrieved_docs = data_source.query(state["question"])
    return {"context": retrieved_docs}

# Compile the graph with branching
graph_builder = StateGraph(State).add_sequence([retrieve, generate])
graph_builder.add_edge(START, "retrieve")
graph = graph_builder.compile()

# Run the graph with branching
```

```
result = graph.invoke({"question": "What is the financial outlook for ZEKO?"})
print(result["answer"])
```

4. HyDe (Hypothetical Document Embeddings)

```
# This example shows how to implement HyDE for better retrieval.

from openai import OpenAI
from pymilvus import MilvusClient
import json
import numpy as np

# Set up OpenAI GPT-3.5
openai_client = OpenAI()

# Connect to Milvus
client = MilvusClient("milvus_demo.db")

# Create a Milvus collection
if client.has_collection(collection_name="hyde_retrieval"):
    client.drop_collection(collection_name="hyde_retrieval")
client.create_collection(
    collection_name="hyde_retrieval",
    dimension=1536
)

# Dummy corpus of documents
corpus = [
    "It usually takes between 30 minutes and two hours to remove a wisdom tooth.",
    "The COVID-19 pandemic has significantly impacted mental health, increasing depressio
    "Humans have used fire for approximately 800,000 years.",
    "Milvus is a cloud based database for vector storage."
]

# Function to get embeddings
def get_embeddings(texts, model="text-embedding-ada-002"):
    response = openai_client.embeddings.create(
        input=texts,
        model=model
    )
    embeddings = [data.embedding for data in response.data]
    return embeddings

# Embed and insert documents into Milvus
vectors = get_embeddings(corpus)
data = [
    {"id": i, "vector": vectors[i], "text": corpus[i]}
    for i in range(len(vectors))
]
client.insert(collection_name="hyde_retrieval", data=data)

# Function to generate a hypothetical document
def generate_hypothetical_document(query):
    response = openai_client.chat.completions.create(
        model="gpt-3.5-turbo-0125",
        messages=[{"role": "system", "content": "Write a document that answers the questi
                  {"role": "user", "content": f"{query}"}],
        max_tokens=100
    )
    return response.choices[^0].message.content

# Function to perform HyDE-based retrieval
def hyde_retrieve(query):
    hypo_doc = generate_hypothetical_document(query)
    hypo_embedding = get_embeddings([hypo_doc])[^0]
    results = client.search(collection_name="hyde_retrieval", data=[hypo_embedding])
    return [corpus[results[i]['id']] for i in range(len(results))]

# Example query
query = "What is Milvus?"
retrieved_docs = hyde_retrieve(query)
print("Retrieved Documents:", retrieved_docs)
```

5. Adaptive RAG

```
# This example demonstrates an adaptive RAG system that adjusts its retrieval strategy ba

# Define a function to analyze query complexity
def analyze_query(query):
    # Placeholder for query analysis logic
    if len(query.split()) &gt; 10:
        return "complex"
    else:
        return "simple"

# Modify the retrieve function to adapt based on query complexity
def retrieve(state: State):
    complexity = analyze_query(state["question"])
    if complexity == "complex":
        # Use a more sophisticated retrieval method
        retrieved_docs = vector_store.similarity_search(state["question"], k=5)
    else:
        # Use a simpler retrieval method
        retrieved_docs = vector_store.similarity_search(state["question"], k=2)
    return {"context": retrieved_docs}

# Compile the graph with adaptive retrieval
graph_builder = StateGraph(State).add_sequence([retrieve, generate])
graph_builder.add_edge(START, "retrieve")
graph = graph_builder.compile()

# Run the graph with adaptive retrieval
result = graph.invoke({"question": "What is the financial outlook for ZEKO?"})
print(result["answer"])
```

6. Corrective RAG (CRAG)

```
# This example shows how to implement CRAG for error detection and correction.

# Define a function to grade document relevance
def grade_document(query, document):
    prompt = f"""Query: {query}
    Document: {document}
    Is this document relevant to the query? Answer with "yes" or "no"."""
    response = completion(
        model="gemini/gemini-1.5-flash",
        messages=[{"content": prompt, "role": "user"}],
        api_key=GEMINI_API_KEY
    )
    answer = response['choices'][^0]['message']['content'].strip().lower()
    return "yes" if "yes" in answer else "no"

# Define a function for supplementary retrieval
def supplementary_retrieval(query):
    search_tool = DuckDuckGoSearchRun()
    web_result = search_tool.invoke(query)
    return web_result

# Define the CRAG pipeline
def corrective_rag(query, top_k=2):
    # Retrieve documents
    results = semantic_search(query, top_k=top_k)
    retrieved_docs = results.get("documents", [])

    # Evaluate relevance
    relevant_docs = []
    for doc in retrieved_docs:
        grade = grade_document(query, doc)
        if grade == "yes":
            relevant_docs.append(doc)

    # If no relevant documents, perform web search
    if not relevant_docs:
        supplementary_doc = supplementary_retrieval(query)
        relevant_docs.append(supplementary_doc)

    # Generate final answer
    context = "\n".join([" ".join(doc) if isinstance(doc, list) else doc for doc in relev
    final_answer = generate_response(query, context)
    return final_answer

# Example usage
query = "What is the insurance for car?"
final_answer = corrective_rag(query)
print("Final Answer:")
print(final_answer)
```

7. Autonomous RAG

```
# This example shows how to implement an autonomous RAG system where the model decides ho

# Define an autonomous agent class
class AutonomousAgent:
    def __init__(self, llm, memory_system):
        self.llm = llm
        self.memory_system = memory_system

    def decide_action(self, query):
        # Placeholder for decision logic
        if "financial" in query.lower():
            return "retrieve_financial_data"
        elif "medical" in query.lower():
            return "retrieve_medical_data"
        else:
            return "generate_response"

    def execute_action(self, action, query):
        if action == "retrieve_financial_data":
            return self.memory_system.query(query)
        elif action == "retrieve_medical_data":
            return self.memory_system.query(query)
        else:
            return self.llm.generate(query)

# Define the autonomous RAG pipeline
def autonomous_rag(query):
    agent = AutonomousAgent(llm, memory_system)
    action = agent.decide_action(query)
    result = agent.execute_action(action, query)
    return result

# Example usage
query = "What is the financial outlook for ZEKO?"
result = autonomous_rag(query)
print(result)
```

8. Agent RAG

```
# This example demonstrates an agent-based RAG system where agents coordinate to process

# Import necessary modules
from swarms import AgentRearrange
from multi_agent_rag.agents import (
    diagnostic_specialist,
    medical_data_extractor,
    patient_care_coordinator,
    specialist_consultant,
    treatment_planner,
)
from multi_agent_rag.memory import LlamaIndexDB
```

```
# Define the agent flow
router = AgentRearrange(
    name="medical-diagnosis-treatment-swarm",
    description="Collaborative medical team for comprehensive patient diagnosis and treat
    max_loops=1,
    agents=[
        medical_data_extractor,
        diagnostic_specialist,
        treatment_planner,
        specialist_consultant,
        patient_care_coordinator,
    ],
    memory_system=LlamaIndexDB(
        data_dir="docs",
        filename_as_id=True,
        recursive=True,
        similarity_top_k=10,
    ),
    flow=f"{medical_data_extractor.agent_name} -&gt; {diagnostic_specialist.agent_name} -
)

# Run the agent-based RAG system
result = router.invoke({"question": "What is the financial outlook for ZEKO?"})
print(result["answer"])
```

These examples provide a basic implementation for each type of RAG architecture. Each can be further customized and expanded based on specific requirements and data sources.

⁂