Generative AI

Module 1: Introduction to LangChain

1. **Overview of LangChain**: Build a simple LLM-powered application for automating customer inquiries.
2. **Setting up LangChain**: Set up a customer support chatbot for live assistance.
3. **Basic Concepts**: Understand how to integrate LangChain with business tools for operational efficiency.

Module 2: Core LangChain Components

4. **Prompt Templates**: Create reusable templates for customer service responses in an e-commerce platform.
5. **Example Selectors**: Use few-shot learning to automate the categorization of incoming customer queries.

Module 3: Working with Chat Models

6. **Chat Models**: Deploy an AI-powered FAQ chatbot for a website to handle common customer questions.
7. **Messages**: Develop a real-time messaging system for personalized client interactions.
8. **Chat History**: Build a sales assistant that retains context across customer interactions.

Module 4: Text Processing and Data Handling

9. **Document Loaders**: Automate document review for processing incoming business proposals.
10. **Text Splitters**: Use text splitters to parse long legal contracts for review and analysis.
11. **Embedding Models**: Implement an intelligent document search feature in a document management system.

Module 5: Data Retrieval and Management

12. **Vector Stores & Retrievers**: Implement a product recommendation engine for an e-commerce platform.
13. **Indexing & Retrieval**: Enhance a knowledge base system for quick retrieval of policies or technical manuals.
14. **Retrieval Augmented Generation (RAG)**: Integrate RAG to provide detailed, context-aware answers using your company's internal knowledge.

Module 6: Customization and Use Cases

15. **Tools and Tool Calling**: Integrate third-party tools (e.g., CRM, payment gateways) to automate workflow processes.

16. **Structured Output**: Automate generating structured invoices from customer orders.
17. **Agents**: Build an intelligent assistant for managing appointments or scheduling meetings based on user needs.

Module 7: Enhancing User Experience

18. **Memory**: Create personalized experiences for returning customers on a retail website.
19. **Chatbots**: Build a customer support chatbot with memory to handle recurring issues efficiently.
20. **Multimodal**: Build an e-commerce search engine that can interpret both image and text queries.

Module 8: Monitoring, Evaluation, and Optimization

21. **Callbacks**: Track customer behavior and trigger targeted marketing actions based on their actions.
22. **Async Programming**: Optimize inventory management systems by asynchronously processing large datasets.
23. **Tracing**: Use tracing to diagnose issues in customer-facing applications.
24. **Evaluation**: Continuously evaluate AI models for accuracy in customer service applications.

Module 9: Advanced Topics

25. **LangChain Expression Language (LCEL)**: Automate complex business workflows, such as contract processing and approval.
26. **Serialization**: Enable the serialization of application states for resuming business processes seamlessly.

Module 10: Real-World Applications

27. **Q&A with RAG**: Implement a Q&A system for internal knowledge bases, helping employees find relevant documents quickly.
28. **Extraction**: Extract and organize data from unstructured reports for financial analysis.
29. **Deploying Chatbots**: Deploy a chatbot for scheduling and managing events for corporate clients.

1. Prompt Templates in Real-Time Business Cases

Overview: A **Prompt Template** in LangChain is used to format a string with user-provided data, typically when the inputs are more straightforward. It is ideal for scenarios where you need to generate consistent outputs based on specific variables.

Use Case in Insurance: Claim Status Notification

In the **insurance industry**, an insurance company might want to notify a customer about the status of their claim. They could use a **PromptTemplate** to generate personalized messages based on the claim's current status.

Business Model:

- **Customer Interaction Model:** Automation of communication channels to keep customers informed about the progress of claims.
- **Goal:** Improve customer satisfaction by sending timely, personalized updates.

Example:

Here, we generate a message to update a customer about the **status of their claim**.

```
from langchain.prompts import PromptTemplate

# Define the prompt template for claim status notification
template = PromptTemplate.from_template("Dear {customer_name}, your claim with
policy number {policy_number} is currently in {claim_status}. We will update you
once the status changes.")

# Input dynamic customer data
customer_name = "John Doe"
policy_number = "XYZ12345"
claim_status = "under review"

# Generate the message using the template
message = template.invoke({"customer_name": customer_name, "policy_number":
policy_number, "claim_status": claim_status})
print(message)
```

Output:

```
Dear John Doe, your claim with policy number XYZ12345 is currently in under
review. We will update you once the status changes.
```

How it works:

- The **PromptTemplate** formats the message dynamically by taking in the **customer's name**, **policy number**, and **claim status**.

- This process allows the insurance company to send personalized messages to customers efficiently.

2. Chat Prompt Templates in Real-Time Business Cases

Overview: A **ChatPromptTemplate** allows for a series of structured messages (e.g., system instructions, user input, assistant responses) to be created, which is more useful for multi-step interactions or conversational interfaces.

Use Case in Retail: Virtual Shopping Assistant

In **retail**, a **virtual shopping assistant** can guide customers through their purchasing process. The assistant could ask questions, recommend products, and follow up with personalized suggestions. A **ChatPromptTemplate** can be used to structure these interactions.

Business Model:

- **Customer Interaction Model:** A conversational chatbot that interacts with customers in real-time to recommend products based on their preferences and behavior.
- **Goal:** Enhance the shopping experience and increase sales by providing personalized product recommendations and offers.

Example:

The virtual assistant can ask customers about their preferences and then recommend a product.

```
from langchain.prompts import ChatPromptTemplate

# Define the prompt template for the virtual shopping assistant
template = ChatPromptTemplate([
    ("system", "You are a helpful virtual shopping assistant."),
    ("user", "I am looking for a {product_type}. What do you recommend?"),
    ("assistant", "Based on your interest in {product_type}, I recommend
{recommendation}. Would you like to know more?"),
])

# Input dynamic product type data
product_type = "laptop"
recommendation = "a high-performance gaming laptop"

# Generate the conversation
```

```
response = template.invoke({"product_type": product_type, "recommendation":
recommendation})
print(response)
```

Output:

```
[('system', 'You are a helpful virtual shopping assistant.'),
 ('user', 'I am looking for a laptop. What do you recommend?'),
 ('assistant', 'Based on your interest in laptop, I recommend a high-performance
gaming laptop. Would you like to know more?')]
```

How it works:

- The assistant guides the customer through a conversation using **structured messages**.
- The system asks for input (e.g., type of product), then provides a personalized recommendation based on the user's query.

3. Messages Placeholder in Real-Time Business Cases

Overview: The **MessagesPlaceholder** allows dynamic insertion of a list of messages into a template. This is useful when dealing with multiple pieces of user-generated content or interactions, such as in multi-turn conversations or when inserting real-time data (like multiple questions or customer responses).

Use Case in Insurance: Customer Support with Multiple Questions

In **insurance**, customers often reach out to support with several questions. A support system can use **MessagesPlaceholder** to insert customer queries dynamically into a message flow. The system could then send back personalized responses, addressing each question in turn.

Business Model:

- **Customer Interaction Model:** Automation of customer service to handle multi-question inquiries in a single conversation.
- **Goal:** Streamline customer service processes and reduce response time by managing multiple customer queries simultaneously.

Example:

A customer may have several questions about their policy, and the system responds accordingly.

```
from langchain.prompts import ChatPromptTemplate, MessagesPlaceholder
from langchain.messages import HumanMessage

# Define the prompt template with placeholder for dynamic messages
template = ChatPromptTemplate([
    ("system", "You are a helpful insurance assistant."),
    MessagesPlaceholder("msgs")
])

# Define multiple customer queries dynamically
customer_queries = [
    HumanMessage(content="What is my policy coverage?"),
    HumanMessage(content="How do I file a claim?")
]

# Generate the conversation
response = template.invoke({"msgs": customer_queries})
print(response)
```

Output:

```
[('system', 'You are a helpful insurance assistant.'),
 ('user', 'What is my policy coverage?'),
 ('user', 'How do I file a claim?')]
```

How it works:

- **MessagesPlaceholder** dynamically inserts a list of customer messages into the conversation.
- This technique is helpful when handling multiple queries, such as when a customer submits several questions at once.

4. Few-Shot Learning in Real-Time Business Cases

Overview: Few-shot learning involves providing a model with a few examples of correct behavior to help it generalize new cases. This is particularly useful when you want to classify or generate responses based on some examples.

Use Case in Retail: Product Review Classification

In the **retail industry**, a company may want to automatically classify product reviews based on their sentiment (e.g., positive, neutral, negative). **Few-shot learning** can help train a model by providing a small set of labeled examples.

Business Model:

- **Customer Interaction Model:** Automating the analysis of customer reviews to quickly identify product sentiment.
- **Goal:** Improve customer satisfaction and provide better recommendations based on review sentiment.

Example:

Classify customer reviews based on sentiment using a few-shot approach.

```
from langchain.prompts import ChatPromptTemplate
from langchain.messages import HumanMessage

# Few-shot examples of sentiment classification
examples = [
    "Review: I love this product! -> Positive",
    "Review: It's okay, nothing special. -> Neutral",
    "Review: Worst purchase I’ve ever made. -> Negative"
]

# Define the prompt template
template = ChatPromptTemplate([
    ("system", "You are a product review classifier."),
    ("user", "Classify the following review: {review}")
])

# Add few-shot examples to the input
review = "Review: The phone is amazing, great battery life! It's my favorite
now."

# Format the input
formatted_input = "\n".join(examples) + f"\nNow, classify the review: {review}"

# Invoke the model
response = template.invoke({"review": formatted_input})
```

```
print(response)
```

Output:

```
Positive
```

How it works:

- The model is provided with a few examples of how to classify reviews based on sentiment.
- The system can then use those examples to classify new reviews in a similar way.

Summary: Real-Time Business Use Cases

1. **Insurance Use Case (PromptTemplate)**:
   a. **Automating Claim Status Notifications**: A **PromptTemplate** helps insurance companies personalize claim status messages based on customer data (name, policy number, claim status).
2. **Retail Use Case (ChatPromptTemplate)**:
   a. **Virtual Shopping Assistant**: A **ChatPromptTemplate** can structure the conversation flow with a user, asking for preferences and then providing product recommendations.
3. **Insurance Use Case (MessagesPlaceholder)**:
   a. **Multi-query Customer Support**: Using **MessagesPlaceholder**, an insurance company can manage multiple customer questions and generate dynamic responses that address all queries.
4. **Retail Use Case (Few-Shot Learning)**:
   a. **Classifying Product Reviews**: **Few-shot learning** allows the system to automatically classify customer reviews based on sentiment (positive, neutral, negative), improving insights and feedback loops for product teams.

Benefits in Business Models:

- **Automation**: Reduces manual effort and improves the speed of interactions (e.g., claim notifications, customer queries, etc.).
- **Personalization**: Ensures that each interaction feels customized to the customer, enhancing user experience and satisfaction.
- **Efficiency**: Helps businesses process large volumes of interactions, whether it's classifying reviews or managing multi-query customer support.

1. Prompt Templates in Business Use Cases

Insurance Use Case: Claim Status Notification

```
from langchain.prompts import PromptTemplate

template = PromptTemplate.from_template("Dear {customer_name}, your claim with
policy number {policy_number} is currently in {claim_status}. We will update you
once the status changes.")

customer_name = "John Doe"
policy_number = "XYZ12345"
claim_status = "under review"

message = template.invoke({"customer_name": customer_name, "policy_number":
policy_number, "claim_status": claim_status})
print(message)
```

Retail Use Case: Personalized Product Recommendation

```
from langchain.prompts import PromptTemplate

template = PromptTemplate.from_template("Based on your interest in
{product_type}, we recommend {recommendation}. Would you like more details?")
product_type = "smartphone"
recommendation = "iPhone 14 Pro"

response = template.invoke({"product_type": product_type, "recommendation":
recommendation})
print(response)
```

2. Few-Shot Examples in Business Use Cases

Insurance Use Case: Classifying Claims by Severity

```
from langchain.prompts import ChatPromptTemplate
```

```
examples = [
    "Claim for a broken arm -> Moderate severity",
    "Claim for a heart attack -> High severity",
    "Claim for a mild cold -> Low severity"
]

template = ChatPromptTemplate([
    ("system", "You are an insurance claims classifier."),
    ("user", "Classify the following claim: {claim}")
])

claim = "Claim for a sprained ankle"
formatted_input = "\n".join(examples) + f"\nNow, classify the claim: {claim}"

response = template.invoke({"claim": formatted_input})
print(response)
```

Retail Use Case: Classifying Product Reviews

```
from langchain.prompts import ChatPromptTemplate

examples = [
    "Review: I love this product! -> Positive",
    "Review: It's okay, nothing special. -> Neutral",
    "Review: Worst purchase I’ve ever made. -> Negative"
]

template = ChatPromptTemplate([
    ("system", "You are a product review classifier."),
    ("user", "Classify the following review: {review}")
])

review = "Review: The phone is amazing, great battery life! It's my favorite
now."
formatted_input = "\n".join(examples) + f"\nNow, classify the review: {review}"

response = template.invoke({"review": formatted_input})
print(response)
```

3. Few-Shot Examples in Chat Models

Insurance Use Case: Handling Multiple Claim Queries

```
from langchain.prompts import ChatPromptTemplate
from langchain.messages import HumanMessage

examples = [
    "Claim for a broken arm -> Moderate severity",
    "Claim for a heart attack -> High severity"
]

template = ChatPromptTemplate([
    ("system", "You are an insurance claims assistant."),
    ("user", "Classify the following claim: {claim}")
])

claims = [
    HumanMessage(content="Claim for a sprained ankle"),
    HumanMessage(content="Claim for a mild headache")
]

formatted_input = "\n".join(examples) + f"\nNow, classify the claims: {claims}"
response = template.invoke({"claims": formatted_input})
print(response)
```

4. Partially Format Prompt Templates in Business Use Cases

Retail Use Case: Customer Order Details

```
from langchain.prompts import PromptTemplate

template = PromptTemplate.from_template("Your order details: {order_id}, Product: 
{product_name}, Quantity: {quantity}")
order_id = "ORD12345"
product_name = "Smart TV"
```

```
quantity = 1

partially_filled = template.partial_format(order_id=order_id)
response = partially_filled.invoke({"product_name": product_name, "quantity":
quantity})
print(response)
```

5. Composing Prompts Together in Business Use Cases

Retail Use Case: Multi-Step Product Recommendation

```
from langchain.prompts import PromptTemplate
from langchain.chains import LLMChain
from langchain.llms import OpenAI

template1 = PromptTemplate.from_template("Given a list of products, recommend a
best-seller: {products}")
template2 = PromptTemplate.from_template("List the key features of the best-
selling product: {product}")

llm = OpenAI(model="text-davinci-003")

chain1 = LLMChain(llm=llm, prompt=template1)
chain2 = LLMChain(llm=llm, prompt=template2)

products = ["Smartphone", "Smart TV", "Laptop"]
best_seller = chain1.run(products=products)

features = chain2.run(product=best_seller)
print("Best-selling product:", best_seller)
print("Key features:", features)
```

2.Document Loaders

Let's explore how **insurance companies** and **retail businesses** can leverage LangChain's document loaders alongside **Large Language Models (LLMs)** to streamline operations, improve efficiency, and gain insights from their data. I'll focus on **real-time use cases** with **complex code examples** that demonstrate how to load and process data (such as PDFs, CSVs, or JSONs), followed by using LLMs to perform analysis or generate reports.

1. Use Case for an Insurance Company: Claims Analysis and Risk Assessment

Problem Statement:

An insurance company needs to process a large volume of claims documents, which are in PDF format. These documents contain unstructured data that needs to be analyzed to identify key information (e.g., claim amount, type of claim, date of claim, etc.). The goal is to automate the extraction of this data and use an LLM to assess risks and provide insights.

Solution:

1. **Load Claims Documents (PDF)**: Load and process the claims data from PDFs.
2. **Use LLMs**: Use an LLM to analyze the extracted text, identify patterns, and assess risks related to the claim (e.g., fraud detection or claim amount prediction).

Code Example:

```
python
Copy code
from langchain.document_loaders import PyPDFLoader
from langchain.prompts import PromptTemplate
from langchain.chat_models import ChatOpenAI
from langchain.chains import LLMChain

# Step 1: Load the Claims Documents (PDF)
pdf_loader = PyPDFLoader("claims_data/claim_123.pdf")
documents = pdf_loader.load()

# Step 2: Extract key information (e.g., claim amount, claim type) using an LLM
# Define a prompt to extract relevant details from the claims document
prompt_template = """
You are an insurance claims analyst. Extract the following details from the
claims document:
- Claim amount
- Type of claim
- Claimant's name
- Claim date
```

```
Here is the claims text:
{document_text}
"""

# Use an LLM to process the claim data
llm = ChatOpenAI(model="gpt-4")

prompt = PromptTemplate(input_variables=["document_text"],
template=prompt_template)
chain = LLMChain(llm=llm, prompt=prompt)

# Extract details for each document
claim_details = []
for doc in documents:
    claim_info = chain.run({"document_text": doc.page_content})
    claim_details.append(claim_info)

# Step 3: Analyze claims for risk assessment (example: fraud detection)
risk_assessment_prompt = """
Given the following claim details, assess the risk of fraud:
- Claim amount: {claim_amount}
- Type of claim: {claim_type}
- Claim date: {claim_date}

Return a risk assessment rating (low, medium, high) and a brief explanation.
"""

# Use the LLM to assess the risk based on the extracted details
for claim in claim_details:
    extracted_data = claim.split("\n")  # Example: Simple data extraction
    claim_amount = extracted_data[0]
    claim_type = extracted_data[1]
    claim_date = extracted_data[2]

    risk_chain = LLMChain(
        llm=llm,
        prompt=PromptTemplate(
            input_variables=["claim_amount", "claim_type", "claim_date"],
            template=risk_assessment_prompt
        )
    )
    risk_score = risk_chain.run({
        "claim_amount": claim_amount,
        "claim_type": claim_type,
        "claim_date": claim_date
    })
    print(f"Risk Assessment for Claim: {risk_score}")
```

Business Impact:

- Automating claims extraction and analysis reduces the time and effort required for claims processing.
- The risk assessment model helps the company identify potential fraudulent claims, improving security and reducing costs.

2. Use Case for a Retail Business: Sales Analysis and Customer Segmentation

Problem Statement:

A retail business wants to analyze customer transaction data stored in **CSV files**. This data includes customer demographics, purchase details, and product categories. The business aims to perform **customer segmentation** to personalize marketing strategies and **predict sales trends**.

Solution:

1. **Load CSV Data**: Load and preprocess the customer transaction data.
2. **Use LLMs**: Apply an LLM to analyze the customer behavior, predict sales trends, and generate targeted marketing strategies.

Code Example:

```
python
Copy code
from langchain.document_loaders import CSVLoader
from langchain.prompts import PromptTemplate
from langchain.chat_models import ChatOpenAI
from langchain.chains import LLMChain

# Step 1: Load the Customer Transaction Data (CSV)
csv_loader = CSVLoader(file_path="customer_data/transactions.csv")
documents = csv_loader.load()

# Step 2: Preprocess Data (simplified example)
customer_data = []
for doc in documents:
    row = doc.metadata  # Assuming metadata contains the columns: ['customer_id',
'age', 'purchase_amount', 'product_category']
```

```
  customer_data.append(row)

# Step 3: Apply LLM for Customer Segmentation and Marketing Strategy
segmentation_prompt = """
Given the following customer transaction data, segment customers into distinct
groups based on their purchasing behavior.
For each segment, suggest a personalized marketing strategy.

Data:
- Customer ID: {customer_id}
- Age: {age}
- Purchase Amount: {purchase_amount}
- Product Category: {product_category}
"""

# Initialize LLM
llm = ChatOpenAI(model="gpt-4")

# Create prompt and chain for customer segmentation
prompt = PromptTemplate(input_variables=["customer_id", "age", "purchase_amount",
"product_category"], template=segmentation_prompt)
chain = LLMChain(llm=llm, prompt=prompt)

# Analyze each customer for segmentation
customer_segments = []
for row in customer_data:
    customer_info = chain.run({
        "customer_id": row['customer_id'],
        "age": row['age'],
        "purchase_amount": row['purchase_amount'],
        "product_category": row['product_category']
    })
    customer_segments.append(customer_info)

# Step 4: Use LLM for Sales Trend Prediction
sales_trend_prompt = """
Based on the following data, predict the sales trend for the next quarter.
- Total sales for the current quarter: {current_sales}
- Most popular product category: {top_product_category}
- Customer segments: {customer_segments}

Provide a prediction for the next quarter's sales growth or decline, and explain
the reasoning.
"""
```

```
# Example: Calculate total sales and top product category (simplified)
current_sales = sum(row['purchase_amount'] for row in customer_data)
top_product_category = max(set([row['product_category'] for row in
customer_data]), key=[row['product_category'] for row in customer_data].count)

# Sales trend prediction
sales_chain = LLMChain(
    llm=llm,
    prompt=PromptTemplate(input_variables=["current_sales",
"top_product_category", "customer_segments"], template=sales_trend_prompt)
)

sales_prediction = sales_chain.run({
    "current_sales": current_sales,
    "top_product_category": top_product_category,
    "customer_segments": customer_segments
})

print(f"Sales Trend Prediction: {sales_prediction}")
```

Business Impact:

- **Customer Segmentation**: Helps the retail business target specific customer groups with tailored marketing campaigns, improving conversion rates.
- **Sales Trend Prediction**: Allows the business to forecast sales trends, which can guide inventory management, staffing decisions, and promotional efforts.

Summary of Key Steps:

1. **Load Data**: Use LangChain's pre-built document loaders to load documents from a variety of sources (PDFs, CSV, etc.).
2. **Preprocess and Analyze Data**: Use Python to process and clean the data.
3. **Use LLMs for Analysis**: Leverage LLMs to perform tasks like information extraction, risk assessment, customer segmentation, and sales prediction.
4. **Generate Insights**: Use LLMs to generate insights, reports, or automated decisions.

These use cases for **insurance** and **retail** businesses demonstrate how LangChain and LLMs can be integrated into business operations to automate data processing, reduce manual effort, and drive better decision-making through AI-powered analysis.

3. Text splitters

Text splitting is crucial for several reasons:

1. **Handling Non-Uniform Document Lengths**: Documents in real-world scenarios can have varying sizes, so splitting ensures uniformity and ease of processing.
2. **Overcoming Model Limitations**: Language models and embedding models have size constraints (e.g., 512 tokens). Splitting ensures that large documents are processed in manageable chunks.
3. **Improving Representation Quality**: Splitting larger documents can improve the accuracy and focus of text representations by reducing the risk of diluting context over long passages.
4. **Optimizing Computational Resources**: Smaller chunks can be processed in parallel, making systems more efficient in terms of memory and speed.

How and When to Use Text Splitting

- **How**: The document is split using different strategies like token-based, semantic-based, or structural splitting depending on the requirements.
- **When**: Use text splitting when processing documents for:
    - Embedding models that have token limits.
    - Information retrieval systems that need efficient indexing.
    - Summarization tasks where logical coherence is required.
    - Large documents (e.g., legal contracts, technical manuals) need to be processed in chunks.

Which Process to Use for Text Splitting

- **Token-based splitting**: When you have a model with token size limitations (e.g., GPT models).
- **Semantic-based splitting**: When the chunks should represent coherent, meaningful content (e.g., customer support knowledge bases).
- **Document-structure-based splitting**: When working with structured documents (e.g., HTML, Markdown, JSON).

1. Token-Based Splitting for Large Legal Documents

In this scenario, legal contracts need to be processed, chunked by token length, and stored in a vector database for retrieval. The chunking process uses overlap to ensure context between splits.

Code Example:

```
from langchain.text_splitters import CharacterTextSplitter
from langchain.embeddings.openai import OpenAIEmbeddings
from langchain.vectorstores import FAISS
import os

# Initialize the tokenizer (e.g., OpenAI's model)
text_splitter = CharacterTextSplitter.from_tiktoken_encoder(
    encoding_name="cl100k_base", chunk_size=512, chunk_overlap=50
)

# Function to split, embed, and store legal contract embeddings
def split_and_store_embeddings(document: str):
    # Split the document based on tokens
    chunks = text_splitter.split_text(document)

    # Generate embeddings for each chunk using OpenAI embeddings
    embeddings = OpenAIEmbeddings()

    # Store embeddings in FAISS (an optimized similarity search library)
    faiss_index = FAISS.from_documents(chunks, embeddings)

    return faiss_index, chunks

# Load the document (e.g., a contract)
with open("large_contract.txt", "r") as file:
    document = file.read()

# Split and store the embeddings
faiss_index, chunks = split_and_store_embeddings(document)

# Example search using FAISS index
def search_document(query: str, faiss_index: FAISS):
    query_embedding = OpenAIEmbeddings().embed_query(query)
    search_results = faiss_index.similarity_search_by_vector(query_embedding,
k=3)
    return search_results

query = "Define the terms of agreement"
```

```
results = search_document(query, faiss_index)
print("Search results:", results)
```

Pros:

- **Simple to implement**: Token-based splitting is straightforward.
- **Consistent chunk sizes**: Ensures chunks are manageable for downstream tasks.
- **Efficient for token-limited models**: Works well with models that have token size constraints.

Cons:

- **Context loss across splits**: Without overlap, chunks might lose some context between them.
- **Token-based approach might be less semantically meaningful**: Some sentences might be cut off mid-way.

2. Recursive Paragraph-Based Splitting for Technical Documentation

For technical documentation, the recursive approach is used to ensure logical coherence by splitting based on paragraphs and sections. This approach is useful for preserving the natural structure of the document.

Code Example:

```
from langchain.text_splitters import RecursiveCharacterTextSplitter
from langchain.embeddings.openai import OpenAIEmbeddings
from langchain.vectorstores import FAISS

# Recursive splitter: will keep paragraphs intact and break on sub-sections if
needed
text_splitter = RecursiveCharacterTextSplitter(chunk_size=400, chunk_overlap=50)

# Function to process and split technical documentation based on natural
structure
def process_technical_document(document: str):
    # Split the document based on paragraphs and sections
    chunks = text_splitter.split_text(document)

    # Generate embeddings for each chunk
    embeddings = OpenAIEmbeddings()
    # Store embeddings using FAISS for fast retrieval
    faiss_index = FAISS.from_documents(chunks, embeddings)

    return faiss_index, chunks

# Load the document (e.g., a technical manual)
with open("technical_manual.txt", "r") as file:
    document = file.read()

# Process and store the document's embeddings
faiss_index, chunks = process_technical_document(document)

# Example search based on user query (e.g., "What is the setup procedure?")
def search_technical_document(query: str, faiss_index: FAISS):
    query_embedding = OpenAIEmbeddings().embed_query(query)
    search_results = faiss_index.similarity_search_by_vector(query_embedding,
k=3)
    return search_results

query = "What is the setup procedure?"
results = search_technical_document(query, faiss_index)
print("Search results:", results)
```

Pros:

- **Preserves logical structure**: Paragraphs and sections remain intact.
- **Improved coherence**: Ensures chunks contain semantically meaningful content by keeping paragraphs together.
- **More flexible**: The approach adapts based on document structure.

Cons:

- **Complexity**: More complex than basic token-based splitting.
- **Chunk size inconsistency**: The chunk sizes might vary based on paragraph lengths.

3. Markdown-Based Splitting for Content Management Systems (CMS)

For content management systems (CMS) where documents are stored in Markdown, we use header-based splitting to preserve content structure, which is useful for indexing and retrieval.

Code Example:

```
import markdown
from langchain.text_splitters import RegexTextSplitter
from langchain.embeddings.openai import OpenAIEmbeddings
from langchain.vectorstores import FAISS

# Regex-based splitter to split by headers in Markdown
header_splitter = RegexTextSplitter(pattern=r"^#{1,6}\s", chunk_size=500,
chunk_overlap=50)

# Function to process and split markdown document based on headers
def process_markdown_document(document: str):
    # Split based on Markdown headers (e.g., #, ##, ###)
    chunks = header_splitter.split_text(document)

    # Generate embeddings for each chunk
    embeddings = OpenAIEmbeddings()

  # Store embeddings using FAISS for efficient retrieval
    faiss_index = FAISS.from_documents(chunks, embeddings)

    return faiss_index, chunks

# Load the Markdown document
with open("markdown_article.md", "r") as file:
    document = file.read()

# Process and store the embeddings
faiss_index, chunks = process_markdown_document(document)

# Example search based on a query (e.g., "What are the installation steps?")
def search_markdown_document(query: str, faiss_index: FAISS):
    query_embedding = OpenAIEmbeddings().embed_query(query)
    search_results = faiss_index.similarity_search_by_vector(query_embedding,
k=3)
    return search_results

query = "What are the installation steps?"
results = search_markdown_document(query, faiss_index)
print("Search results:", results)
```

Pros:

- **Maintains document structure**: Preserves headers and sections, making it ideal for CMS or wiki-like content.
- **Efficient retrieval**: With proper structure-based splitting, it's easier to index and retrieve relevant sections.
- **Good for Markdown content**: Perfect for scenarios where documents are structured with headers.

Cons:

- **Header dependency**: The splitting logic heavily relies on well-structured Markdown headers.
- **Fixed chunk sizes**: The chunks are still limited by a character or token size.

4. Semantic-Based Splitting with Sliding Window Approach (For Product Manuals)

For product manuals or knowledge bases, semantic splitting ensures that the content is chunked based on meaningful breaks, such as changes in topic. This improves coherence for tasks like summarization or customer support query handling.

Code Example:

```
from langchain.embeddings.openai import OpenAIEmbeddings
from langchain.text_splitters import RecursiveCharacterTextSplitter
from langchain.vectorstores import FAISS
import numpy as np

# Initialize embedding model
embeddings = OpenAIEmbeddings()

# Function to compute semantic differences between consecutive chunks
def semantic_split(document: str, chunk_size: int, overlap: int):
    # Use a sliding window to split text semantically
    splitter = RecursiveCharacterTextSplitter(chunk_size=chunk_size,
chunk_overlap=overlap)
    chunks = splitter.split_text(document)

    # Generate embeddings for each chunk
    chunk_embeddings = [embeddings.embed_document(chunk) for chunk in chunks]

    # Compare embeddings to detect breaks in meaning
    semantic_chunks = []
    current_chunk = []
    previous_embedding = None

    for i, embedding in enumerate(chunk_embeddings):
        if previous_embedding is not None:
            similarity = np.dot(previous_embedding, embedding) /
(np.linalg.norm(previous_embedding) * np.linalg.norm(embedding))
            if similarity < 0.85:  # Significant semantic difference detected
                semantic_chunks.append(' '.join(current_chunk))
                current_chunk = []
        current_chunk.append(chunks[i])
        previous_embedding = embedding

    # Append the last chunk
    semantic_chunks.append(' '.join(current_chunk))

    return semantic_chunks

# Load the document (e.g., a product manual)
with open("product_manual.txt", "r") as file:
    document = file.read()

# Split the document semantically
semantic_chunks = semantic_split(document, chunk_size=
```

500, overlap=50)

Store embeddings using FAISS for efficient retrieval

faiss_index = FAISS.from_documents(semantic_chunks, embeddings)

Example search using semantic chunks

def search_semantic_document(query: str, faiss_index: FAISS): query_embedding = embeddings.embed_query(query) search_results = faiss_index.similarity_search_by_vector(query_embedding, k=3) return search_results

query = "How to assemble the product?" results = search_semantic_document(query, faiss_index) print("Search results:", results)

```
 Pros:
- Semantic coherence: Chunks are split based on the meaning, ensuring better
understanding.
- Ideal for deep document analysis: Suitable for use cases like summarization,
customer support, and FAQs.

 Cons:
- Complexity: Requires embedding models and additional logic for detecting
semantic breaks.
- More computationally intensive: Needs embedding models for every chunk and
comparison between consecutive chunks.

---

 Conclusion

In real-world use cases, text splitting plays a critical role in document
processing, making the content manageable for models, ensuring efficient
retrieval, and maintaining logical coherence. Each approach—token-based,
recursive, header-based, and semantic splitting—has its own strengths and
limitations, and the choice of which to use depends on the type of document and
the specific use case:
```

```
- Token-based splitting is simple and efficient but may lose context.
- Recursive splitting is ideal for structured documents but may have varying
chunk sizes.
- Header-based splitting is great for Markdown or structured content but depends
on the document's formatting.
- Semantic splitting provides the most natural breaks but is computationally
intensive and more complex to implement.

Each of these techniques helps to break down large and complex documents into
manageable chunks, optimizing them for NLP tasks, knowledge retrieval, and
document analysis.
```

4.Embeddings

1) Why We Are Doing and Importance for That

- **Embeddings** are used to convert text into numerical representations (vectors), allowing machines to understand, compare, and search for semantic meaning efficiently.
- They enable **semantic search**: going beyond keyword matching, embeddings compare meanings.
- Embeddings improve **retrieval systems** by transforming documents and queries into vectors, where similarity is measured.
- They help reduce the complexity and computational cost of text-based tasks, enabling faster, more scalable solutions.
- Essential in building advanced AI applications such as **question-answering systems**, **recommendation systems**, and **content-based filtering**.

2) When We Can Do

- **When building search engines**: Embeddings allow for semantic search, where the relevance of results is based on meaning, not just keyword matching.
- **When performing text clustering or classification**: Embeddings allow grouping similar documents or content together.
- **For knowledge retrieval systems**: Embedding models help retrieve the most semantically relevant information from large document sets.
- **When personalizing user experiences**: Embeddings can be used to personalize content recommendations by understanding user preferences.
- **For integrating large language models**: Embedding models are essential when combining with LLMs to make intelligent, context-aware decisions.

3) Number of Procedures and Methods to Follow

1. **Choosing an Embedding Model Provider**: Select from a variety of embedding models (OpenAI, HuggingFace, etc.).
2. **Embedding the Documents**: Use LangChain's `embed_documents` method to convert multiple documents into vectors.
3. **Embedding the Query**: Use LangChain's `embed_query` method to convert a single query into a vector.
4. **Measuring Similarity**: Choose a similarity metric (e.g., cosine similarity, dot product, or Euclidean distance) to compare the query vector with document vectors.
5. **Search or Retrieval**: After embedding, use the similarity measure to retrieve the most relevant documents.
6. **Fine-Tuning**: Optionally fine-tune the embeddings to your specific domain if necessary.
7. **Scaling and Optimizing**: Consider caching results and optimizing computations for real-time large-scale use cases.

4) Generate Code for Real-Time and Advanced Business Cases

Example Business Case: Health Sector

In healthcare, embedding models can be used for medical literature search, patient data analysis, or drug discovery.

```
from langchain_openai import OpenAIEmbeddings
from langchain.vectorstores import FAISS

# Initialize OpenAI embedding model
embeddings_model = OpenAIEmbeddings()

# Sample medical documents (e.g., patient records, medical literature)
documents = [
    "The patient is suffering from acute bronchitis and needs an inhaler.",
    "The new vaccine has been shown to be effective in preventing flu symptoms.",
    "Patients with diabetes should monitor their blood sugar levels regularly."
]

# Embed documents
embedded_documents = embeddings_model.embed_documents(documents)

# Store embeddings in a vector store for fast similarity search
vector_store = FAISS.from_embeddings(embedded_documents)

# Query about medical symptoms or treatments
query = "What is the best treatment for acute bronchitis?"
```

```
query_embedding = embeddings_model.embed_query(query)

# Search for most relevant documents
search_results = vector_store.similarity_search(query_embedding)
print(search_results)
```

- **Health Scenario**: A patient asks a system about the best treatment for a disease, and the system searches through a vast database of medical literature to find the most relevant treatment options using embeddings and similarity search.

Example Business Case: Retail Sector

For personalized recommendations in retail, embeddings can be used to match products with customer preferences.

```
from langchain_openai import OpenAIEmbeddings
from langchain.vectorstores import FAISS

# Sample product descriptions
products = [
    "A sleek smartphone with a high-resolution camera.",
    "A comfortable sofa with modern design and ergonomic support.",
    "Wireless earbuds with noise cancellation and long battery life."
]

# Embed product descriptions
embeddings_model = OpenAIEmbeddings()
product_embeddings = embeddings_model.embed_documents(products)

# Store embeddings in FAISS index
product_store = FAISS.from_embeddings(product_embeddings)

# User query (e.g., based on product preferences)
query = "I need a new phone with a great camera"
query_embedding = embeddings_model.embed_query(query)

# Search for most relevant products
recommended_products = product_store.similarity_search(query_embedding)
print(recommended_products)
```

- **Retail Scenario**: A customer searches for products like "smartphones with a great camera," and the system uses embeddings to retrieve the most relevant product descriptions based on semantic understanding.

Example Business Case: Insurance Sector

In the insurance industry, embeddings can be used to match clients with suitable insurance policies or assess claims.

```
from langchain_openai import OpenAIEmbeddings
from langchain.vectorstores import FAISS

# Example insurance policy descriptions
policies = [
    "Health insurance policy with low premiums and wide coverage for family healthcare.",
    "Life insurance with high coverage for beneficiaries in case of accidents.",
    "Auto insurance offering protection against accidents, theft, and third-party
liability."
]

# Embed insurance policies
embeddings_model = OpenAIEmbeddings()
policy_embeddings = embeddings_model.embed_documents(policies)

# Store embeddings in vector database
policy_store = FAISS.from_embeddings(policy_embeddings)

# Client query (e.g., looking for family health coverage)
query = "I need affordable health insurance for my family"
query_embedding = embeddings_model.embed_query(query)

# Retrieve the best matching insurance policies
matching_policies = policy_store.similarity_search(query_embedding)
print(matching_policies)
```

- **Insurance Scenario**: A client queries about the best affordable health insurance for their family. The system uses embeddings to match the query with the most relevant insurance policies,

1. How to Embed Text Data

```
from langchain_openai import OpenAIEmbeddings

embeddings_model = OpenAIEmbeddings()

documents = [
    "The patient is suffering from acute bronchitis and needs an inhaler.",
    "A new vaccine has been shown to be effective in preventing flu symptoms.",
    "Patients with diabetes should monitor their blood sugar levels regularly."
]

document_embeddings = embeddings_model.embed_documents(documents)

print(len(document_embeddings))
print(len(document_embeddings[0]))
```

2. How to Cache Embedding Results

```
import joblib
from langchain_openai import OpenAIEmbeddings

embeddings_model = OpenAIEmbeddings()

def embed_and_cache(documents, cache_filename="document_embeddings.pkl"):
    try:
        embeddings = joblib.load(cache_filename)
        print("Loaded cached embeddings.")
    except FileNotFoundError:
        embeddings = embeddings_model.embed_documents(documents)
        joblib.dump(embeddings, cache_filename)
        print("Calculated and cached embeddings.")
    return embeddings

documents = [
    "The patient is suffering from acute bronchitis and needs an inhaler.",
    "A new vaccine has been shown to be effective in preventing flu symptoms.",
    "Patients with diabetes should monitor their blood sugar levels regularly."
]
```

```
document_embeddings = embed_and_cache(documents)
```

3. How to Create a Custom Embeddings Class

```
from langchain.embeddings.base import Embeddings

class CustomEmbeddings(Embeddings):
    def __init__(self, model_name: str):
        self.model_name = model_name

    def embed_documents(self, documents: list) -> list:
        embeddings = []
        for document in documents:
            embedding = self._custom_embedding_logic(document)
            embeddings.append(embedding)
        return embeddings

    def embed_query(self, query: str) -> list:
        return self._custom_embedding_logic(query)

    def _custom_embedding_logic(self, text: str):
        return [0.0] * 1536

custom_embeddings = CustomEmbeddings(model_name="custom-model")

documents = [
    "The patient is suffering from acute bronchitis and needs an inhaler.",
    "A new vaccine has been shown to be effective in preventing flu symptoms."
]
custom_embeddings_result = custom_embeddings.embed_documents(documents)

query = "What is the best treatment for acute bronchitis?"
custom_query_embedding = custom_embeddings.embed_query(query)

print(custom_embeddings_result)
```

5) Pros and Cons of the Above Process

Pros:

- **Efficiency**: Embedding models allow for fast retrieval based on semantic meaning, reducing time spent on keyword-based searches.
- **Scalability**: Embedding models and vector databases like FAISS allow handling large datasets, making them suitable for big businesses with large document collections.
- **Accuracy**: Embeddings improve search accuracy, as they take the context and meaning into account, rather than relying solely on exact keyword matches.
- **Personalization**: Embedding models can be used to provide personalized recommendations based on user preferences, improving customer experience.
- **Versatility**: Can be applied across a variety of domains (e.g., healthcare, retail, insurance) for search, recommendation, and information retrieval tasks.

Cons:

- **Complexity**: Setting up and optimizing embeddings for large-scale real-time systems can be complex, requiring expertise in machine learning and natural language processing.
- **Computational Cost**: Embedding models, particularly LLMs, can require significant computational resources, especially when dealing with large-scale data.
- **Latency**: In some real-time applications, the time it takes to embed documents and perform similarity searches can introduce delays, especially in high-volume systems.
- **Data Privacy**: In sensitive sectors like healthcare or insurance, embedding models must be carefully implemented to avoid exposing sensitive information during processing.

This explanation lays out the theoretical foundation for working with embeddings in LangChain, demonstrates real-world business applications, and discusses the pros and cons of the approach.

Vector Stores & Retrivers

1) Why We Are Doing and Importance for That

- **Vector Stores** enable the efficient storage and retrieval of data using embeddings, which are fixed-length vector representations of text, images, audio, and other data types.
- **Semantic Search**: Vector stores allow searching based on semantic similarity rather than exact keyword matches.

- **Unstructured Data**: With vector stores, large unstructured data sets (like documents, images, or customer reviews) can be effectively indexed and queried.
- **Real-Time Applications**: In industries like healthcare, retail, and insurance, vector stores allow for scalable and fast retrieval of relevant information to assist with decision-making.
- **Advanced Use Cases**: In business models like healthcare, vector stores enable advanced text-based querying over patient data, medical literature, etc., while in retail, it powers recommendation systems. For insurance, it helps in quickly finding relevant policies and claims data.

2) When We Can Do This

- **When dealing with large volumes of unstructured data**: Text, images, or audio that need to be indexed and retrieved efficiently.
- **When semantic search is required**: When keyword search is insufficient, and you need to retrieve information based on meaning and context.
- **When metadata filtering is necessary**: For refining search results based on specific attributes, such as document type or source.
- **When you require fast retrieval**: For systems that must operate in real-time, like recommendation engines or customer support systems.

3) Number of Procedures and Methods We Need to Follow for the Above Things

1. **Initializing Vector Store**: Choose a vector store implementation (e.g., in-memory, Pinecone, FAISS, etc.) and initialize it with the chosen embedding model.
   a. Method: `InMemoryVectorStore`, `Pinecone`, `FAISS`, etc.
   b. Embedding model integration: `SomeEmbeddingModel()`
2. **Adding Documents**: Documents or data points (text, image metadata) must be added to the vector store, where each document is embedded into a vector representation.
   a. Method: `add_documents`
   b. Documents need to be in `Document` format, containing content and metadata.
3. **Deleting Documents**: Delete documents when no longer needed or for data pruning purposes.
   a. Method: `delete_documents`
4. **Searching for Similar Documents**: Perform a semantic search by embedding a query and comparing it with stored document embeddings.
   a. Method: `similarity_search`
   b. Similarity metrics: Cosine Similarity, Euclidean Distance, or Dot Product.
5. **Advanced Search and Retrieval**: Implement techniques like **Maximal Marginal Relevance (MMR)** for diversifying results or **Hybrid Search** for combining keyword and semantic searches.

    a. Method: Use search parameters like k (number of results) or `filter` (metadata filtering).
6. **Metadata Filtering**: Filter search results based on metadata attributes, which is essential for narrowing down the search scope.
    a. Method: `similarity_search` with metadata filters.

4) Real-Time Business Case Example and Advanced Code Generation

Scenario: Healthcare - Patient Data Retrieval

In the healthcare industry, we need to index medical records, patient information, and treatment guidelines. The goal is to enable healthcare professionals to search for relevant documents based on symptoms, diagnoses, or treatments quickly.

Real-Time Code:

```
from langchain_core.vectorstores import InMemoryVectorStore
from langchain_openai import OpenAIEmbeddings
from langchain_core.documents import Document

# Initialize embedding model and vector store
embedding_model = OpenAIEmbeddings()
vector_store = InMemoryVectorStore(embedding=embedding_model)

# Sample healthcare documents
document_1 = Document(
    page_content="Patient diagnosed with acute bronchitis, prescribed inhaler and
cough syrup.",
    metadata={"condition": "bronchitis", "treatment": "inhaler"}
)

document_2 = Document(
    page_content="Patient with diabetes, advised regular blood sugar level
monitoring and exercise.",
    metadata={"condition": "diabetes", "treatment": "blood sugar monitoring"}
)

# Add documents to vector store
vector_store.add_documents(documents=[document_1, document_2], ids=["doc1",
"doc2"])
```

```
# Search for similar documents based on query
query = "What treatment is best for bronchitis?"
results = vector_store.similarity_search(query, k=2)

# Display results
for doc in results:
    print(doc.page_content)
```

Scenario: Retail - Product Recommendation System

In retail, vector stores are used to store product descriptions and user queries. When a user asks for product recommendations (e.g., a "laptop with good battery life"), the system searches for products with similar descriptions based on embeddings.

```
from langchain_core.vectorstores import InMemoryVectorStore
from langchain_openai import OpenAIEmbeddings
from langchain_core.documents import Document

# Initialize embedding model and vector store
embedding_model = OpenAIEmbeddings()
vector_store = InMemoryVectorStore(embedding=embedding_model)

# Sample retail documents
document_1 = Document(
    page_content="Laptop with a 15-inch display and 12-hour battery life.",
    metadata={"category": "electronics", "type": "laptop"}
)

document_2 = Document(
    page_content="Smartphone with a 6-inch screen and 10-hour battery life.",
    metadata={"category": "electronics", "type": "smartphone"}
)

# Add documents to vector store
vector_store.add_documents(documents=[document_1, document_2], ids=["prod1",
"prod2"])

# Search for similar products based on query
```

```
query = "laptop with long battery life"
results = vector_store.similarity_search(query, k=1)

# Display results
for doc in results:
    print(doc.page_content)
```

Scenario: Insurance - Policy Retrieval

In the insurance industry, vector stores can help quickly retrieve policies or claims data based on a user's query about coverage or claim details.

```
from langchain_core.vectorstores import InMemoryVectorStore
from langchain_openai import OpenAIEmbeddings
from langchain_core.documents import Document

# Initialize embedding model and vector store
embedding_model = OpenAIEmbeddings()
vector_store = InMemoryVectorStore(embedding=embedding_model)

# Sample insurance documents
document_1 = Document(
    page_content="Home insurance policy covering fire, theft, and natural
disasters.",
    metadata={"type": "home", "coverage": "fire, theft, natural disasters"}
)

document_2 = Document(
    page_content="Car insurance policy covering accidents, vandalism, and third-
party liability.",
    metadata={"type": "car", "coverage": "accidents, vandalism, liability"}
)

# Add documents to vector store
vector_store.add_documents(documents=[document_1, document_2], ids=["policy1",
"policy2"])

# Search for similar documents based on query
query = "home insurance for fire damage"
```

```
results = vector_store.similarity_search(query, k=1)

# Display results
for doc in results:
    print(doc.page_content)
```

Real-Time Business Scenarios & Code Examples for Vector Stores and Retrievers

Business Case 1: Health Insurance Document Search & Retrieval

Scenario: A health insurance company wants to quickly retrieve customer queries related to coverage, policy details, and medical claims from a large database of insurance policy documents.

1. Use a Vector Store to Retrieve Data

```
from langchain_core.vectorstores import InMemoryVectorStore
from langchain_openai import OpenAIEmbeddings
from langchain_core.documents import Document

embedding_model = OpenAIEmbeddings()
vector_store = InMemoryVectorStore(embedding=embedding_model)

documents = [
    Document(page_content="Coverage details for dental health insurance",
metadata={"category": "dental"}),
    Document(page_content="Procedure for submitting a medical claim",
metadata={"category": "claims"})
]

vector_store.add_documents(documents=documents)

query = "How do I file a medical claim?"
retrieved_docs = vector_store.similarity_search(query)
```

2. Generate Multiple Queries to Retrieve Data for

```
queries = ["What does my dental coverage include?", "How do I submit a claim?"]
retrieved_docs = [vector_store.similarity_search(query) for query in queries]
```

3. Use Contextual Compression to Compress the Data Retrieved

```
from langchain.text_summarizers import Summarizer

summarizer = Summarizer()
compressed_docs = [summarizer.summarize(doc.page_content) for doc in
retrieved_docs[0]]
```

4. Write a Custom Retriever Class

```
from langchain_core.retrievers import Retriever

class CustomInsuranceRetriever(Retriever):
    def retrieve(self, query):
        query_embedding = embedding_model.embed_query(query)
        return vector_store.similarity_search(query_embedding)
```

5. Add Similarity Scores to Retriever Results

```
import numpy as np

def cosine_similarity(vec1, vec2):
    dot_product = np.dot(vec1, vec2)
    norm_vec1 = np.linalg.norm(vec1)
    norm_vec2 = np.linalg.norm(vec2)
    return dot_product / (norm_vec1 * norm_vec2)

retrieved_docs_with_scores = [
    (doc, cosine_similarity(query_embedding, doc.embedding)) for doc in
retrieved_docs[0]
]
```

6. Combine the Results from Multiple Retrievers

```
query1 = "What is the process to submit a claim?"
query2 = "What is the dental coverage included in the policy?"

retrieved_docs_1 = vector_store.similarity_search(query1)
retrieved_docs_2 = vector_store.similarity_search(query2)

combined_results = list(set(retrieved_docs_1 + retrieved_docs_2))
```

7. Reorder Retrieved Results to Mitigate "Lost in the Middle" Effect

```
sorted_results = sorted(combined_results, key=lambda doc: doc.relevance_score,
reverse=True)
```

8. Generate Multiple Embeddings Per Document

```
chunks = ["Coverage details for dental health insurance", "Procedure for
submitting a medical claim"]
embedding_results = [embedding_model.embed_query(chunk) for chunk in chunks]
```

9. Retrieve the Whole Document for a Chunk

```
query_chunk = "Procedure for submitting a medical claim"
retrieved_chunk = vector_store.similarity_search(query_chunk)
full_document = [doc for doc in documents if doc.page_content ==
retrieved_chunk[0].page_content]
```

10. Generate Metadata Filters

```
metadata_filter = {"category": "claims"}
filtered_results = vector_store.similarity_search(query, filter=metadata_filter)
```

11. Create a Time-Weighted Retriever

```
from datetime import datetime


def time_weighted_similarity(doc, query, current_time):
    time_diff = (current_time - doc.metadata["timestamp"]).days
    weight = max(1 / (time_diff + 1), 0.1)
    return weight * cosine_similarity(doc.embedding, query.embedding)

current_time = datetime.now()
time_weighted_docs = [
    (doc, time_weighted_similarity(doc, query, current_time)) for doc in
retrieved_docs[0]
]
```

12. Use Hybrid Vector and Keyword Retrieval

```
keyword_results = vector_store.keyword_search("medical claim process")
vector_results = vector_store.similarity_search("medical claim process")
combined_results = list(set(keyword_results + vector_results))
```

Business Case 2: Retail Product Search

Scenario: A large retail store wants to provide customers with product recommendations based on user queries (e.g., product specifications, features).

1. Use a Vector Store to Retrieve Data

```
from langchain_core.vectorstores import InMemoryVectorStore
from langchain_openai import OpenAIEmbeddings
from langchain_core.documents import Document

embedding_model = OpenAIEmbeddings()
vector_store = InMemoryVectorStore(embedding=embedding_model)

documents = [
    Document(page_content="Smartphone with 8GB RAM and 128GB storage",
metadata={"category": "smartphones"}),
    Document(page_content="Laptop with 16GB RAM and 512GB SSD",
metadata={"category": "laptops"})
]

vector_store.add_documents(documents=documents)
```

```
query = "I am looking for a laptop with 16GB RAM"
retrieved_docs = vector_store.similarity_search(query)
```

2. Generate Multiple Queries to Retrieve Data for

```
queries = ["Smartphones with large storage", "Laptops with good performance for
gaming"]
retrieved_docs = [vector_store.similarity_search(query) for query in queries]
```

3. Use Contextual Compression to Compress the Data Retrieved

```
summarizer = Summarizer()
compressed_docs = [summarizer.summarize(doc.page_content) for doc in
retrieved_docs[0]]
```

4. Write a Custom Retriever Class

```
class CustomProductRetriever(Retriever):
    def retrieve(self, query):
        query_embedding = embedding_model.embed_query(query)
        return vector_store.similarity_search(query_embedding)
```

5. Add Similarity Scores to Retriever Results

```
retrieved_docs_with_scores = [
    (doc, cosine_similarity(query_embedding, doc.embedding)) for doc in
retrieved_docs[0]
]
```

6. Combine the Results from Multiple Retrievers

```
retrieved_docs_1 = vector_store.similarity_search("Best laptops for gaming")
retrieved_docs_2 = vector_store.similarity_search("Smartphones with 128GB
storage")
combined_results = list(set(retrieved_docs_1 + retrieved_docs_2))
```

7. Reorder Retrieved Results to Mitigate "Lost in the Middle" Effect

```
sorted_results = sorted(combined_results, key=lambda doc: doc.relevance_score,
reverse=True)
```

8. Generate Multiple Embeddings Per Document

```
chunks = ["Smartphone with 8GB RAM", "Laptop with 16GB RAM"]
embedding_results = [embedding_model.embed_query(chunk) for chunk in chunks]
```

9. Retrieve the Whole Document for a Chunk

```
retrieved_chunk = vector_store.similarity_search("Smartphone with 8GB RAM")
full_document = [doc for doc in documents if doc.page_content ==
retrieved_chunk[0].page_content]
```

10. Generate Metadata Filters

```
metadata_filter = {"category": "laptops"}
filtered_results = vector_store.similarity_search("Best laptops for gaming",
filter=metadata_filter)
```

11. Create a Time-Weighted Retriever

```
current_time = datetime.now()

def time_weighted_similarity(doc, query, current_time):
    time_diff = (current_time - doc.metadata["timestamp"]).days
    weight = max(1 / (time_diff + 1), 0.1)
    return weight * cosine_similarity(doc.embedding, query.embedding)

time_weighted_docs = [
    (doc, time_weighted_similarity(doc, query, current_time)) for doc in
retrieved_docs[0]
]
```

12. Use Hybrid Vector and Keyword Retrieval

```
keyword_results = vector_store.keyword_search("laptops with good performance")
vector_results = vector_store.similarity_search("laptops with good performance")
combined_results = list(set(keyword_results + vector_results))
```

5) Pros and Cons of the Above Process

Pros:

- **Semantic Search**: Enables retrieval of information based on meaning rather than exact keyword matches, improving search accuracy.
- **Real-Time Retrieval**: Vector stores support fast and scalable search, making them suitable for high-performance, real-time applications.
- **Flexible and Scalable**: Works with large datasets of unstructured data, allowing for diverse use cases across industries like healthcare, retail, and insurance.
- **Metadata Filtering**: Allows for structured queries, refining search results based on metadata attributes, which helps narrow down large data sets.

Cons:

- **Computational Cost**: Embedding large datasets can be computationally expensive, especially for high-dimensional embeddings.
- **Complex Setup**: Setting up vector stores and embedding models may require infrastructure and fine-tuning for optimal performance.
- **Storage Requirements**: Storing large volumes of vectors may require significant storage capacity, especially for high-dimensional vectors.
- **Search Precision**: While vector stores provide high-speed retrieval, search results might not always be as precise as keyword-based systems, requiring additional filtering or re-ranking.

Indexing and Retrivals

1) Why We Are Doing Indexing and Its Importance

- **Purpose**: Indexing ensures that the vector store remains synchronized with the underlying data source, which can be updated or changed over time.

- **Importance**: It allows for real-time access to the most recent information and ensures that the vector store is accurate and up-to-date. Without proper indexing, stale or outdated data could be returned in search queries, leading to incorrect results.
- **Use Cases**: Indexing is crucial in scenarios where the data source is dynamic, such as health records, retail inventory, or insurance policies, where frequent updates (additions, deletions, or modifications) occur.

2) When We Can Do Indexing

- **Data Source Changes**: Indexing is necessary whenever new data is added, deleted, or updated in the underlying data source.
- **Scheduled Updates**: If the vector store is regularly updated (e.g., daily or weekly) based on changes to the data, indexing should be performed on a schedule.
- **When Performance Issues Arise**: If the vector store is slowing down due to the increasing volume of data, re-indexing may improve performance and retrieval efficiency.

3) Procedures and Methods to Follow for Indexing

- **Step 1: Identify Data Changes**

Monitor changes in the data source (such as new documents added, old documents deleted, or documents updated).

- **Step 2: Extract Relevant Data**

When changes are identified, extract the new, deleted, or updated data that needs to be re-indexed.

- **Step 3: Remove Stale Data**

Remove the outdated documents from the vector store to maintain accuracy and prevent stale information from being retrieved.

- **Step 4: Update or Add New Data**

Add new or updated data to the vector store using the appropriate embedding model to generate embeddings.

- **Step 5: Ensure Consistency**

Ensure that the vector store reflects the data source by checking the successful addition of documents and the removal of outdated ones.

4) Generate Code for Real-Time Business Case Scenarios

Scenario 1: Health Insurance Document Indexing

Use Case: A health insurance company must keep its vector store up-to-date with new insurance policy documents, claims, and regulations.

Procedure:

```
from langchain_core.vectorstores import InMemoryVectorStore
from langchain_openai import OpenAIEmbeddings
from langchain_core.documents import Document

# Initialize vector store with embedding model
embedding_model = OpenAIEmbeddings()
vector_store = InMemoryVectorStore(embedding=embedding_model)

# Original documents
documents = [
    Document(page_content="Insurance policy details for outpatient care",
metadata={"category": "policy", "id": "policy_1"}),
    Document(page_content="Process to file a medical claim",
metadata={"category": "claims", "id": "claim_1"})
]

# Add initial documents to vector store
vector_store.add_documents(documents=documents)

# Scenario: Updating data
new_documents = [
    Document(page_content="New policy details for outpatient care coverage",
metadata={"category": "policy", "id": "policy_2"})
]

# Remove outdated documents
vector_store.delete_documents(ids=["policy_1"])

# Re-index new/updated documents
```

```
vector_store.add_documents(documents=new_documents)
```

Scenario 2: Retail Product Catalog Indexing

Use Case: A retail store needs to keep the product catalog in sync with new arrivals and sales updates.

Procedure:

```
from langchain_core.vectorstores import InMemoryVectorStore
from langchain_openai import OpenAIEmbeddings
from langchain_core.documents import Document

# Initialize vector store
embedding_model = OpenAIEmbeddings()
vector_store = InMemoryVectorStore(embedding=embedding_model)

# Original product catalog
documents = [
    Document(page_content="Smartphone with 8GB RAM and 128GB storage",
metadata={"category": "smartphones", "id": "product_1"}),
    Document(page_content="Laptop with 16GB RAM and 512GB SSD",
metadata={"category": "laptops", "id": "product_2"})
]

# Add initial products to vector store
vector_store.add_documents(documents=documents)

# Scenario: Updating product catalog
updated_products = [
    Document(page_content="New Laptop with 16GB RAM and 1TB SSD",
metadata={"category": "laptops", "id": "product_2"})
]

# Remove outdated products
vector_store.delete_documents(ids=["product_1"])

# Re-index updated products
```

```
vector_store.add_documents(documents=updated_products)
```

Scenario 3: Insurance Claim Data Indexing

Use Case: An insurance company needs to track claims and provide search results based on historical claims data.

Procedure:

```
from langchain_core.vectorstores import InMemoryVectorStore
from langchain_openai import OpenAIEmbeddings
from langchain_core.documents import Document

# Initialize vector store
embedding_model = OpenAIEmbeddings()
vector_store = InMemoryVectorStore(embedding=embedding_model)

# Initial claims data
claims_documents = [
    Document(page_content="Claim for car accident filed on January 5, 2024",
metadata={"category": "claims", "id": "claim_1"}),
    Document(page_content="Claim for water damage filed on February 10, 2024",
metadata={"category": "claims", "id": "claim_2"})
]

# Add claims documents to vector store
vector_store.add_documents(documents=claims_documents)

# Scenario: Claim data updates
updated_claims = [
    Document(page_content="Updated claim for car accident filed on January 5,
2024", metadata={"category": "claims", "id": "claim_1"})
]

# Remove outdated claims
vector_store.delete_documents(ids=["claim_2"])

# Re-index updated claims
```

```
vector_store.add_documents(documents=updated_claims)
```

5) Pros and Cons of Indexing Process Using LLMs and LangChain

Pros:

- **Real-Time Updates**: Ensures that the vector store always has the most recent and relevant data for querying.
- **Scalability**: With tools like LangChain, it is easy to scale and index large datasets efficiently.
- **Customization**: LangChain provides flexibility to integrate with various vector stores and supports custom retrievers, making the indexing process highly adaptable to specific business needs.
- **Accurate Search**: Keeping the vector store in sync ensures that search results are based on the latest information, improving overall user satisfaction.

Cons:

- **High Computational Cost**: Re-indexing, especially with large datasets, can be resource-intensive, requiring significant computation and storage.
- **Data Consistency Issues**: If the indexing process is not carefully managed, inconsistencies between the data source and the vector store can lead to incorrect or outdated search results.
- **Performance Overhead**: Frequent re-indexing can create performance overhead, especially in real-time business scenarios, where data updates may occur constantly.
- **Complexity**: Managing the synchronization process between data sources and vector stores can add complexity, particularly in highly dynamic environments such as retail, health, and insurance sectors.

Retrival Agumented Genaration

1) Why We Are Doing Retrieval Augmented Generation (RAG) and Its Importance

- **Purpose**: RAG combines language models with external knowledge sources to overcome the limitation of fixed internal knowledge. It allows models to provide real-time, domain-specific, and factually grounded responses.
- **Importance**: It enables AI systems to offer up-to-date, accurate information by querying external data sources, bridging the gap between model knowledge and dynamic, evolving datasets.

- **Use Cases**: Critical in industries like healthcare, retail, and insurance, where fast, accurate, and context-specific information is needed for decision-making and customer support.

2) When We Can Use RAG

- **Outdated Internal Knowledge**: When models are unable to answer questions based on recent events or new knowledge that wasn't part of their training.
- **Domain-Specific Requirements**: In industries like healthcare or finance, when the model needs access to very specific data (e.g., medical guidelines, insurance policies) to provide valuable insights.
- **Minimizing Hallucinations**: When the language model generates potentially inaccurate or invented information due to lack of sufficient data.
- **Cost-Effective Knowledge Integration**: When updating a model through fine-tuning is costly or inefficient, RAG provides a more scalable, less resource-heavy alternative.

3) Procedures and Methods to Follow for Implementing RAG

- **Step 1: Define the Retrieval System**
  - Set up an external knowledge base (e.g., databases, documents, APIs) to fetch relevant data.
- **Step 2: Formulate Query Input**
  - Structure the input query that will be sent to the retrieval system to ensure that it is specific and aligned with the external knowledge sources.
- **Step 3: Search for Relevant Information**
  - Use the retrieval system to query relevant information based on the input query.
- **Step 4: Integrate Retrieved Information**
  - Pass the retrieved information to the language model as context within the model's prompt.
- **Step 5: Model Response Generation**
  - Generate a response by leveraging the context provided by the retrieval system, ensuring that the answer is grounded in the retrieved data.
- **Step 6: Return Answer**
  - Return the generated response to the user, ensuring that it is concise and based on factual, retrieved data.

4) Generate Code for Real-Time Business Case Scenarios

Scenario 1: Health Industry Knowledge Retrieval (RAG for Medical Advice)

Use Case: A health insurance company needs an AI model to retrieve medical guidelines and provide specific answers to questions about insurance coverage for treatments.

```
from langchain_openai import ChatOpenAI
from langchain_core.messages import SystemMessage, HumanMessage
from langchain.vectorstores import Pinecone
from langchain.vectorstores import FAISS

# Define a retriever (FAISS or Pinecone can be used as the knowledge base)
retriever = FAISS.load_local("medical_guidelines_index")

# System prompt for grounding response
system_prompt = """You are a medical assistant.
Use the following pieces of retrieved context to answer the patient's question.
If you don't know the answer, say 'I don't know.'
Context: {context}:"""

# Question
question = """What are the coverage guidelines for outpatient treatment for
diabetes?"""

# Retrieve relevant documents based on the query
docs = retriever.similarity_search(question)

# Combine retrieved documents into a single string
docs_text = "".join(d.page_content for d in docs)

# Format the system prompt with the retrieved information
system_prompt_fmt = system_prompt.format(context=docs_text)

# Initialize the language model (using GPT-4 in this case)
model = ChatOpenAI(model="gpt-4", temperature=0)

# Generate response
response = model.invoke([SystemMessage(content=system_prompt_fmt),
                         HumanMessage(content=question)])

# Return the generated response
print(response)
```

Scenario 2: Retail Product Query (RAG for Product Recommendation)

Use Case: A retail store AI needs to recommend products to a customer based on specific queries related to features and inventory.

```
from langchain_openai import ChatOpenAI
from langchain_core.messages import SystemMessage, HumanMessage
from langchain.vectorstores import Pinecone

# Define a retriever (Pinecone or FAISS stores product-related data)
retriever = Pinecone.load_local("retail_products_index")

# Define system prompt
system_prompt = """You are a product recommendation assistant.
Use the following retrieved product details to suggest the best product.
If you cannot find a suitable match, say 'No matching products available.'
Context: {context}:"""

# Customer query
question = """Can you recommend a laptop with 16GB RAM and a 1TB SSD?"""

# Retrieve relevant product information
docs = retriever.similarity_search(question)

# Combine the documents into one context string
docs_text = "".join(d.page_content for d in docs)

# Format the system prompt with the retrieved data
system_prompt_fmt = system_prompt.format(context=docs_text)

# Initialize the language model (e.g., GPT-4)
model = ChatOpenAI(model="gpt-4", temperature=0)

# Generate response
response = model.invoke([SystemMessage(content=system_prompt_fmt),
                         HumanMessage(content=question)])

# Print the response
```

```
print(response)
```

Scenario 3: Insurance Claim Verification (RAG for Verifying Claims)

Use Case: An insurance company uses RAG to verify claims by retrieving relevant policies or historical data from the knowledge base.

```
from langchain_openai import ChatOpenAI
from langchain_core.messages import SystemMessage, HumanMessage
from langchain.vectorstores import FAISS

# Define a retriever (FAISS for historical insurance claim data)
retriever = FAISS.load_local("insurance_claims_index")

# Define system prompt for claim verification
system_prompt = """You are an insurance assistant.
Use the following documents to verify if this claim is valid.
If the claim is not valid, say 'Claim not valid'.
Context: {context}:"""

# Claim query
question = """Does the claim for water damage meet the policy criteria for
reimbursement?"""

# Retrieve relevant documents (e.g., policy, claims data)
docs = retriever.similarity_search(question)

# Combine retrieved documents into a context string
docs_text = "".join(d.page_content for d in docs)

# Format the system prompt
system_prompt_fmt = system_prompt.format(context=docs_text)

# Initialize the language model (e.g., GPT-4)
model = ChatOpenAI(model="gpt-4", temperature=0)

# Generate response
response = model.invoke([SystemMessage(content=system_prompt_fmt),
                                   HumanMessage(content=question)])

# Return response
print(response)
```

Business Case Scenarios and Code for Retrieval Augmented Generation (RAG)

RAG (Retrieval Augmented Generation) is a powerful approach where language models like GPT are augmented with external knowledge from various databases, knowledge bases, or document repositories. This enables them to generate responses that are more informed and contextually relevant. Below are the scenarios for Healthcare, Retail, and Insurance, with complex code examples for each.

1. How to Add Chat History

In this business case, we'll look at a **Healthcare Chatbot** that tracks user queries and responds based on a medical knowledge base.

Code Example: Healthcare Chatbot with Chat History

```
from langchain.chat_models import ChatOpenAI
from langchain.chains import ConversationChain
from langchain.memory import ConversationBufferMemory
from langchain.vectorstores import FAISS
from langchain.embeddings.openai import OpenAIEmbeddings
from langchain.prompts import ChatPromptTemplate

# Initialize the embeddings and FAISS index
embeddings = OpenAIEmbeddings()
faiss_index = FAISS.load_local("medical_faiss_index", embeddings)

# Initialize the memory to track conversation history
memory = ConversationBufferMemory(memory_key="chat_history")

# Create the chat model
chat_model = ChatOpenAI(model="gpt-4")

# Create the conversation chain
```

```
conversation = ConversationChain(
    llm=chat_model,
    memory=memory,
    retriever=faiss_index.as_retriever(search_type="similarity_search")
)

# Function to query the chatbot
def query_health_chatbot(user_input):
    response = conversation.predict(input=user_input)
    return response

# Example chat interaction
user_input = "What are the symptoms of diabetes?"
response = query_health_chatbot(user_input)
print(f"Chatbot Response: {response}")
```

Explanation:

- **Purpose**: This chatbot allows users to ask medical questions, keeping track of the conversation history.
- **How it works**: The `ConversationBufferMemory` class stores all prior user inputs and model responses. The `ChatOpenAI` model uses this history to provide context for generating accurate responses.
- **Business Case**: Healthcare providers could use this for patient interactions, where the model needs to remember previous inquiries about symptoms, medications, or diagnoses.

2. How to Stream Responses

Streaming allows for responses to be generated progressively, providing real-time feedback, which is important in environments like customer support or medical consultations.

Code Example: Streaming Responses for Retail Customer Support

```
from langchain.chat_models import ChatOpenAI
from langchain.callbacks import StreamingStdOutCallback
from langchain.prompts import ChatPromptTemplate
```

```
# Initialize the chat model with streaming capability
chat_model = ChatOpenAI(model="gpt-4", streaming=True,
callbacks=[StreamingStdOutCallback()])

# Create the prompt template for product queries
prompt_template = ChatPromptTemplate.from_messages([
    ("system", "You are a customer support agent for a retail store."),
    ("user", "{question}")
])

# Function to stream a response to a retail query
def query_retail_support(query):
    prompt = prompt_template.format_prompt(question=query)
    response = chat_model.predict_messages(messages=prompt.messages)
    return response

# Example query
query = "Can you help me find a laptop with good battery life?"
response = query_retail_support(query)
print(f"Retail Support Response: {response}")
```

Explanation:

- **Purpose**: Real-time interaction with customers for retail product queries. Responses are streamed as they are generated.
- **How it works**: The `StreamingStdOutCallback()` streams responses back as they are generated by the model. Useful in applications where customers expect fast replies.
- **Business Case**: Retail customer service systems can implement this for faster, more interactive customer support, especially in e-commerce scenarios.

3. How to Return Sources

In RAG, the external data source (e.g., product descriptions or medical records) is critical. Returning sources ensures that users understand where the information is coming from.

Code Example: Insurance Claim Assistance with Source Retrieval

```
from langchain.prompts import PromptTemplate
from langchain.vectorstores import FAISS
from langchain.embeddings.openai import OpenAIEmbeddings
from langchain.chains import RetrievalQA
from langchain.chat_models import ChatOpenAI

# Initialize the retriever for insurance claim data
embeddings = OpenAIEmbeddings()
faiss_index = FAISS.load_local("insurance_claim_faiss", embeddings)

# Create the chat model
chat_model = ChatOpenAI(model="gpt-4")

# Create the RetrievalQA chain with sources returned
qa_chain = RetrievalQA.from_chain_type(
    llm=chat_model,
    retriever=faiss_index.as_retriever(search_type="similarity_search"),
    return_source_documents=True
)

# Function to process the insurance claim query and return sources
def query_insurance_claim(query):
    result = qa_chain.run(query)
    return result['result'], result['source_documents']

# Example query
query = "What is the procedure to file a claim for a car accident?"
response, sources = query_insurance_claim(query)
print(f"Response: {response}")
print(f"Sources: {sources}")
```

Explanation:

- **Purpose**: In the insurance industry, it's vital for customers to know where information comes from, such as specific insurance policies or legal documents.
- **How it works**: The `RetrievalQA` chain retrieves the most relevant documents based on the query and returns the sources along with the response.

- **Business Case**: An insurance company's FAQ system or chatbot can respond to queries while showing the exact policy documents or guidelines the response was derived from.

4. How to Return Citations

For domains like healthcare, legal, and research, citations are critical for ensuring trustworthiness and compliance.

Code Example: Healthcare Research Assistance with Citations

```
from langchain.prompts import PromptTemplate
from langchain.vectorstores import FAISS
from langchain.embeddings.openai import OpenAIEmbeddings
from langchain.chains import RetrievalQA
from langchain.chat_models import ChatOpenAI

# Load the vectorstore with medical research papers
embeddings = OpenAIEmbeddings()
faiss_index = FAISS.load_local("medical_research_faiss", embeddings)

# Initialize the chat model
chat_model = ChatOpenAI(model="gpt-4")

# Create the RetrievalQA chain for healthcare research
qa_chain = RetrievalQA.from_chain_type(
    llm=chat_model,
    retriever=faiss_index.as_retriever(search_type="similarity_search"),
    return_source_documents=True
)

# Function to handle medical queries with citations
def query_medical_research(query):
    result = qa_chain.run(query)
    citations = [doc.metadata['citation'] for doc in result['source_documents']]
    return result['result'], citations

# Example query
query = "What are the latest findings on Alzheimer's treatments?"
```

```
response, citations = query_medical_research(query)
print(f"Response: {response}")
print(f"Citations: {citations}")
```

Explanation:

- **Purpose**: In a healthcare research application, it's necessary to return citations for any medical advice or findings provided.
- **How it works**: The `RetrievalQA` chain retrieves relevant documents, and the citations are extracted from the document metadata.
- **Business Case**: Healthcare professionals or researchers can use this feature to provide evidence-backed responses to queries on treatment protocols or medical innovations.

5. How to Do Per-User Retrieval

Per-user retrieval allows the system to customize responses based on the individual's history or preferences.

Code Example: Personalized Retail Recommendations

```
from langchain.vectorstores import FAISS
from langchain.embeddings.openai import OpenAIEmbeddings
from langchain.chat_models import ChatOpenAI
from langchain.chains import RetrievalQA

# Initialize the user's purchase history and embeddings
user_embeddings = OpenAIEmbeddings()
faiss_index = FAISS.load_local("user_purchase_history_faiss", user_embeddings)

# Create the personalized retrieval chat model
chat_model = ChatOpenAI(model="gpt-4")

# Create the personalized QA chain
qa_chain = RetrievalQA.from_chain_type(
    llm=chat_model,
    retriever=faiss_index.as_retriever(search_type="similarity_search"),
    return_source_documents=True
)

# Function to retrieve personalized product recommendations
def get_personalized_recommendations(user_id, query):
    user_vector_store = FAISS.load_local(f"user_{user_id}_history_faiss",
user_embeddings)
    personalized_qa_chain = RetrievalQA.from_chain_type(
        llm=chat_model,

retriever=user_vector_store.as_retriever(search_type="similarity_search"),
        return_source_documents=True
    )
    result = personalized_qa_chain.run(query)
    return result['result'], result['source_documents']

# Example query for a user with ID 12345
query = "What are some good winter jackets for outdoor activities?"
response, sources = get_personalized_recommendations("12345", query)
print(f"Personalized Recommendation: {response}")
print(f"Sources: {sources}")
```

Explanation:

- **Purpose**: This code is designed for personalized product recommendations based on user purchase history.
- **How it works**: The user's previous interactions (purchase history, preferences) are stored in a user-specific FAISS index, and retrieval is customized for each user.
- **Business Case**: Retailers can implement this to provide personalized shopping experiences, where customers receive recommendations based on their past purchases or browsing behavior.

Conclusion

The code examples provided demonstrate how RAG can be implemented across various business sectors such as healthcare, retail, and insurance. Each case uses advanced features like chat history, streaming, returning sources, citations, and per-user retrieval to create powerful, personalized, and data-driven systems. By leveraging external knowledge bases and combining them with LLMs, businesses can deliver more relevant, accurate, and actionable insights in real-time.

5) Pros and Cons of Retrieval Augmented Generation (RAG)

Pros:

- **Up-to-date Information**: RAG ensures that the model can provide real-time, accurate information from dynamic data sources.
- **Domain-Specific Expertise**: It allows for more specific, tailored responses by using domain-specific knowledge bases.
- **Reduces Hallucinations**: Since responses are grounded in real, retrieved data, RAG reduces the likelihood of the model generating false or invented information.
- **Cost-Efficient**: Compared to fine-tuning a model, RAG is a more cost-effective way to integrate external knowledge and keep it up-to-date.

Cons:

- **Dependence on External Data**: RAG requires an external knowledge base or retriever to work effectively, and its performance depends on the quality and scope of the data.
- **Latency Issues**: The retrieval process may introduce latency, especially when querying large external data sources, which could affect real-time use cases.
- **Complexity**: The setup for RAG can be complex, requiring an efficient retrieval system and management of external knowledge sources.
- **Data Privacy Concerns**: When using external knowledge bases, sensitive data might be exposed during the retrieval process, raising potential privacy and security issues.

Tools and Tool Calling

1) Why We Are Doing Tools and Tool Calling, and Their Importance

- **Purpose of Tools**: Tools in AI models, particularly in language models, provide extended functionalities by integrating additional capabilities (e.g., accessing external databases, performing complex computations, invoking external APIs).
- **Importance of Tools**: Tools enhance the AI model's capabilities beyond fixed knowledge, allowing dynamic and real-time interactions. This is crucial in industries that require up-to-date, domain-specific, and complex information processing like healthcare, insurance, and retail.
- **Purpose of Tool Calling**: Tool calling allows models to interact programmatically with these external tools, ensuring that the language model can retrieve real-time data, manipulate information, or trigger actions in external systems.

2) When We Can Use Tools and Tool Calling

- **Real-Time Data Needs**: When a model requires up-to-date, specific, or dynamic data beyond its training set (e.g., querying a live database or an API).
- **Complex Problem-Solving**: When a solution requires more than just natural language processing, such as performing calculations, querying specialized knowledge sources, or interacting with external systems.
- **Business-Specific Requirements**: Industries such as healthcare (e.g., querying a medical database), retail (e.g., checking product availability), or insurance (e.g., verifying claims) often require real-time, complex data processing.
- **Automation of Tasks**: When AI-driven models need to automate certain tasks, like making API calls to third-party services, interacting with systems, or triggering actions (e.g., booking appointments or creating insurance claims).

3) Procedures and Methods to Follow for Tools and Tool Calling

- **Step 1: Define the Tool Schema**
  - The tool schema specifies the name, description, and arguments the function accepts.
- **Step 2: Set Up Tool Integration**
  - Implement the tool with the language model (for example, connecting it to external APIs, databases, or computational services).
- **Step 3: Define Tool Behavior**
  - Specify how the tool should respond to inputs and which external resource or action it should invoke.
- **Step 4: Incorporate Tool Calling**
  - Integrate tool calls into the language model's decision-making process. This can be done via API calls, function executions, or interactions with other services.
- **Step 5: Handle Inputs and Outputs**
  - Ensure that the inputs to the tool (e.g., queries, data) are properly formatted and passed. Similarly, the outputs should be handled appropriately and used by the model for generating the final response.
- **Step 6: Test and Iterate**
  - Test the system with different scenarios and iterate to improve the effectiveness of tool integration, ensuring that the tool calling enhances the model's capabilities.

4) Generate Code for Real-Time Business Case Scenarios

Scenario 1: Health Industry Tool for Real-Time Medical Query (Tool Calling for External Database)

Use Case: A health AI system that queries a live medical database to provide up-to-date drug interaction information.

```
from langchain_openai import ChatOpenAI
from langchain_core.messages import SystemMessage, HumanMessage
from langchain.tools import Tool
import requests

# Define tool schema for querying the medical database
class DrugInteractionTool(Tool):
    name = "drug_interaction_tool"
    description = "Queries a medical database to find drug interactions."
    arguments = {
        "drug_name": "string",
    }

    def call(self, drug_name):
        # API request to a medical database
        response =
requests.get(f"http://medicaldatabase.com/interactions/{drug_name}")
        return response.json()

# Initialize model
model = ChatOpenAI(model="gpt-4", temperature=0)

# Define the system prompt for the AI to call the tool
system_prompt = """You are a medical assistant.
Use the following tool to find information on drug interactions:
{tool}
Context: {context}:"""

# Define a query and retrieve drug interaction information
drug_name = "aspirin"
tool = DrugInteractionTool()
interaction_info = tool.call(drug_name)

# Format the response
system_prompt_fmt = system_prompt.format(tool=tool, context=interaction_info)

# Model generates a response using the retrieved data
```

```
response = model.invoke([SystemMessage(content=system_prompt_fmt),
                         HumanMessage(content=f"Tell me about the interactions of
{drug_name}")])

# Output the model response
print(response)
```

Scenario 2: Retail AI Tool for Product Availability Check (Tool Calling for Inventory System)

Use Case: A retail AI system querying inventory for product availability and recommending alternatives.

```
from langchain_openai import ChatOpenAI
from langchain_core.messages import SystemMessage, HumanMessage
from langchain.tools import Tool
import requests

# Tool for querying retail inventory system
class InventoryCheckTool(Tool):
    name = "inventory_check_tool"
    description = "Queries the inventory system for product availability."
    arguments = {
        "product_id": "string",
    }

    def call(self, product_id):
        # API request to the inventory system
        response = requests.get(f"http://inventorysystem.com/check/{product_id}")
        return response.json()

# Initialize model
model = ChatOpenAI(model="gpt-4", temperature=0)

# Define system prompt for the AI to call the tool
system_prompt = """You are a retail assistant.
Use the following tool to check product availability:
{tool}
Context: {context}:"""
```

```
# Query product availability
product_id = "laptop_1234"
tool = InventoryCheckTool()
availability_info = tool.call(product_id)

# Format the response
system_prompt_fmt = system_prompt.format(tool=tool, context=availability_info)

# Generate response from the model
response = model.invoke([SystemMessage(content=system_prompt_fmt),
                         HumanMessage(content=f"Is product {product_id}
available?")])

# Output the response
print(response)
```

Scenario 3: Insurance Claim Tool for Verification (Tool Calling for Policy Check)

Use Case: An AI system checking whether an insurance claim is valid based on a customer's policy.

```
from langchain_openai import ChatOpenAI
from langchain_core.messages import SystemMessage, HumanMessage
from langchain.tools import Tool
import requests

# Tool for querying insurance policy data
class InsuranceClaimCheckTool(Tool):
    name = "insurance_claim_check_tool"
    description = "Queries the insurance system to validate a claim."
    arguments = {
        "claim_id": "string",
    }

    def call(self, claim_id):
        # API request to the insurance database
        response = requests.get(f"http://insurancesystem.com/claim/{claim_id}")
        return response.json()
```

```
# Initialize model
model = ChatOpenAI(model="gpt-4", temperature=0)

# Define system prompt for the AI to call the tool
system_prompt = """You are an insurance assistant.
Use the following tool to validate a claim:
{tool}
Context: {context}:"""

# Query for claim validation
claim_id = "claim_5678"
tool = InsuranceClaimCheckTool()
claim_info = tool.call(claim_id)

# Format the response
system_prompt_fmt = system_prompt.format(tool=tool, context=claim_info)

# Generate response from the model
response = model.invoke([SystemMessage(content=system_prompt_fmt),
                         HumanMessage(content=f"Is claim {claim_id} valid?")])

# Output the response
print(response)
```

Here is a breakdown of the processes mentioned in the list with code examples and explanations for each of them, with the integration of LangChain tools in real-time business cases like health, retail, and insurance.

1) How to Create Tools

Creating a custom tool in LangChain involves defining a function that can be invoked by the model. The tool schema describes the name, description, and arguments, while the function defines the actual behavior of the tool.

Example: Health Scenario (Drug Interaction Checker Tool)

```
from langchain.tools import Tool
import requests

class DrugInteractionTool(Tool):
    name = "drug_interaction_tool"
    description = "Checks for potential interactions between drugs using a
medical database"
    arguments = {
        "drug_name": "string",
    }

    def call(self, drug_name):
        # Mock API call to an external medical database for drug interactions
        response =
requests.get(f"https://api.medicaldb.com/interactions/{drug_name}")
        if response.status_code == 200:
            return response.json()  # Return interactions
        else:
            return {"error": "Could not fetch data from database."}
```

2) How to Use Built-in Tools and Toolkits

LangChain provides built-in tools like APIs for databases, or predefined actions for specific tasks. You can use these tools without having to code them from scratch.

Example: Retail Scenario (Inventory Check Tool)

```
from langchain.tools import Tool
import requests

class InventoryCheckTool(Tool):
    name = "inventory_check_tool"
    description = "Checks inventory levels of a product"
    arguments = {
        "product_id": "string",
    }

    def call(self, product_id):
        response =
requests.get(f"https://api.retailstore.com/inventory/{product_id}")
        if response.status_code == 200:
            return response.json()
        else:
            return {"error": "Could not fetch product information."}
```

3) How to Use Chat Models to Call Tools

You can integrate tools within the LangChain chat model system to allow the model to invoke them based on queries.

Example: Insurance Scenario (Insurance Claim Check)

```
from langchain_openai import ChatOpenAI
from langchain_core.messages import SystemMessage, HumanMessage
from langchain.tools import Tool
import requests

class InsuranceClaimCheckTool(Tool):
    name = "insurance_claim_check_tool"
    description = "Validates an insurance claim based on claim ID"
    arguments = {
        "claim_id": "string",
    }

    def call(self, claim_id):
        response = requests.get(f"https://api.insurance.com/check/{claim_id}")
        if response.status_code == 200:
            return response.json()
        else:
            return {"error": "Could not validate claim."}

# Model setup
model = ChatOpenAI(model="gpt-4", temperature=0)

# Define system prompt for the AI to call the tool
system_prompt = """You are an insurance assistant.
Use the following tool to validate an insurance claim:
{tool}
Context: {context}:"""

claim_id = "claim_5678"
```

```
tool = InsuranceClaimCheckTool()
claim_info = tool.call(claim_id)

system_prompt_fmt = system_prompt.format(tool=tool, context=claim_info)

response = model.invoke([SystemMessage(content=system_prompt_fmt),
                         HumanMessage(content=f"Is claim {claim_id} valid?")])

print(response)
```

4) How to Pass Tool Outputs to Chat Models

The output of the tool (like data from an API call) is passed to the model's prompt to provide context for the model's response.

Example: Health Scenario (Drug Interaction Query)

```
# Drug interaction tool setup
tool = DrugInteractionTool()

# Simulating a query for drug interactions
drug_name = "aspirin"
interaction_info = tool.call(drug_name)

# Passing the tool output to a model
system_prompt = """You are a medical assistant.
Use the following context to provide a concise answer about drug interactions:
Context: {context}"""

system_prompt_fmt = system_prompt.format(context=interaction_info)

response = model.invoke([SystemMessage(content=system_prompt_fmt),
                         HumanMessage(content=f"What are the interactions of
{drug_name}?")])

print(response)
```

5) How to Pass Runtime Values to Tools

Passing runtime values involves using dynamic data (user inputs or real-time information) that can be passed into tools during execution.

Example: Retail Scenario (Real-time Product Query)

```
product_id = input("Enter product ID to check availability: ")

tool = InventoryCheckTool()
availability_info = tool.call(product_id)

# Respond with dynamic output
system_prompt = f"Product availability for {product_id}: {availability_info}"
response = model.invoke([SystemMessage(content=system_prompt),
                         HumanMessage(content=f"Is product {product_id}
available?")])

print(response)
```

6) How to Add a Human-in-the-loop for Tools

This feature ensures that human intervention is possible during tool execution, especially for critical decisions or cases requiring approval.

Example: Insurance Scenario (Human Review for Claims)

```
def review_claim_manually(claim_id):
    # Simulate manual review by an insurance agent
    print(f"Manual review required for claim {claim_id}.")
    decision = input("Approve or deny claim? (approve/deny): ")
    return decision

claim_id = "claim_5678"
tool = InsuranceClaimCheckTool()
claim_info = tool.call(claim_id)

# Human-in-the-loop for decision-making
if claim_info['status'] != 'approved':
    claim_decision = review_claim_manually(claim_id)
else:
    claim_decision = "Claim approved automatically."

print(f"Claim Decision: {claim_decision}")
```

7) How to Handle Tool Errors

Handling errors is critical to ensure that tools don’t fail unexpectedly and can provide meaningful feedback when something goes wrong.

Example: Health Scenario (Error Handling in Drug Interaction Check)

```
class DrugInteractionTool(Tool):
    name = "drug_interaction_tool"
    description = "Checks for potential interactions between drugs using a
medical database"
    arguments = {
        "drug_name": "string",
    }

    def call(self, drug_name):
        try:
            response =
requests.get(f"https://api.medicaldb.com/interactions/{drug_name}")
            response.raise_for_status()
            return response.json()  # Return interactions
        except requests.exceptions.RequestException as e:
            return {"error": f"Error while fetching drug interactions: {str(e)}"}
```

8) How to Force Models to Call a Tool

You can explicitly instruct the model to use a tool to generate an output, overriding the default behavior.

```
tool = DrugInteractionTool()
drug_name = "ibuprofen"
interaction_info = tool.call(drug_name)
```

```
# Model forced to call tool
model = ChatOpenAI(model="gpt-4", temperature=0)

system_prompt = """You are a medical assistant.
Use the following context to provide a concise answer:
{tool}
Context: {context}"""

system_prompt_fmt = system_prompt.format(tool=tool, context=interaction_info)

response = model.invoke([SystemMessage(content=system_prompt_fmt),
                         HumanMessage(content=f"Tell me about interactions of
{drug_name}")])

print(response)
```

9) How to Disable Parallel Tool Calling

In scenarios where you want to ensure that tool invocations are made sequentially to avoid concurrency issues, you can disable parallel tool calling.

```
# Disable parallel tool calling by sequentially invoking each tool
tool_1 = InventoryCheckTool()
tool_2 = InsuranceClaimCheckTool()

product_info = tool_1.call("product_1234")
claim_info = tool_2.call("claim_5678")

# Process sequentially to avoid parallel tool calling issues
```

10) How to Access the RunnableConfig from a Tool

RunnableConfig allows you to manage configurations of the tool when running the tool.

```
# Access and configure the tool for specific tasks
tool_config = tool.get_config()
```

```
# Print tool configuration
print(tool_config)
```

11) How to Stream Events from a Tool

Stream event functionality is used to track the progress of tool execution in real-time.

```
# Define the tool to stream events during execution
def stream_events_from_tool():
    tool = InventoryCheckTool()
    for event in tool.stream():
        print(f"Event: {event}")
```

12) How to Return Artifacts from a Tool

You can return additional artifacts like logs, files, or generated results that the model can use later.

```
# Returning artifacts
class ArtifactTool(Tool):
    name = "artifact_tool"
    description = "Generates artifacts during tool execution."
    arguments = {
        "input_data": "string",
    }

    def call(self, input_data):
        artifact = f"Processed: {input_data}"
        # Return artifact
        return artifact
```

13) How to Convert Runnables to Tools

A "Runnable" in LangChain refers to a tool that can execute some process and return a result. You can convert runnables to tools for easy integration.

```
from langchain.runnables import Runnable
from langchain.tools import Tool

# Convert runnable to tool
class RunnableTool(Tool):
    name = "runnable_tool"
    description = "Executes a task and returns a result."

    def call(self, input_data):
        runnable = Runnable(input_data)
        result = runnable.execute()
        return result
```

14) How to Add Ad-hoc Tool Calling Capability to Models

To add ad-hoc tool calling, models can dynamically choose which tool to call based on inputs.

```
# Dynamic tool calling
def call_tool_based_on_input(input_data):
    if "product" in input_data:
        tool = InventoryCheckTool()
    elif "claim" in input_data:
        tool = InsuranceClaimCheckTool()
    else:
        return "Invalid input"

    return tool.call(input_data)
```

15) How to Pass in Runtime Secrets

Runtime secrets, such as API keys, can be injected securely into tools at runtime.

```
import os

# Set API key as environment variable
os.environ['API_KEY'] = 'your_api_key_here'
```

```
# Pass runtime secret (API key) into tool
class APIKeyTool(Tool):
    name = "api_key_tool"
    description = "Tool that uses runtime secrets."
    arguments = {
        "query": "string",
    }

    def call(self, query):
        api_key = os.getenv('API_KEY')
        response =
requests.get(f"https://api.example.com/data?key={api_key}&query={query}")
        return response.json()
```

```
These are just a few examples of how to implement various LangChain tool
functionalities to solve real-time business problems across various sectors.
```

5) Pros and Cons of Tools and Tool Calling

Pros:

- **Extends Model Capabilities**: Tools provide functionalities that are outside the model's knowledge, allowing the AI to retrieve real-time information, interact with external systems, and perform actions.
- **Real-Time Data Integration**: Using tools allows models to incorporate up-to-date data, which is critical for industries like healthcare, insurance, and retail.
- **Complex Task Automation**: Tools enable the automation of complex tasks that involve external systems, such as querying databases, calling APIs, and executing computations, making AI systems more versatile.
- **Customization**: Tools can be tailored to specific business needs, such as querying a custom inventory system, verifying insurance policies, or accessing medical databases.

Cons:

- **Dependency on External Systems**: Tool calling relies heavily on the availability and quality of the external systems and APIs, which can introduce points of failure.

- **Latency**: Calls to external tools (e.g., databases, APIs) may introduce latency, affecting real-time responsiveness, especially in mission-critical applications.
- **Complexity**: Integrating tools into an AI system requires careful design and management of tool interactions, which can complicate the development process.
- **Security Risks**: Integrating external tools and systems might introduce security concerns, especially when handling sensitive data like health information, insurance policies, or customer information.

Agents

1) Why We Are Doing It and the Importance

- **Reason**: Agents are designed to enable language models to perform complex tasks by making decisions about which actions to take. By allowing the model to choose and sequence actions dynamically, it can interact with tools, external data sources, and perform logical tasks beyond simple response generation.
- **Importance**: It allows for task automation, decision-making, and complex workflows in real-time. Agents bridge the gap between language models and practical problem-solving in domains such as healthcare, insurance, retail, and more.
    - Increased automation in business processes.
    - Dynamic, context-aware decision-making.
    - Handling tasks that require interaction with multiple external systems (e.g., APIs, databases).

2) When We Can Do It

- **Use Cases**:
    - When an agent needs to make decisions based on the context or environment.
    - When multiple sequential actions are required, and decision-making needs to adapt.
    - In cases where actions involve interacting with external APIs, databases, or systems dynamically.
    - Scenarios that require complex workflows across multiple domains or systems.
- **Timing**: Agents are useful when building AI systems that must handle long-term decision-making or when the problem involves various external actions such as querying external resources, updating databases, or communicating with other systems.

3) Procedures and Methods to Follow

1. **Define Actions and Tools**: First, define the set of actions the agent can perform, such as querying databases, calling APIs, or interacting with tools.
2. **Create a Language Model-Based Decision-Making System**: Use a language model to decide the next best action based on the input and current context.
3. **Create an Action Loop**: Use a loop where the agent chooses an action, performs it, and returns the result. The agent repeats this process based on the outcomes of previous actions.
4. **Integrate Tools**: Link the tools (databases, APIs, and others) with the agent so that it can interact with them to gather or act on data.
5. **Handle Outcomes and Context**: Ensure that the agent processes the results of its actions and adapts its subsequent actions accordingly.
6. **Implement Termination or Completion Logic**: Define when the agent should stop acting, typically after a goal is achieved or when no further actions are possible.
7. **Test and Evaluate**: Continuously evaluate the agent's performance and adjust its decision-making rules as needed.

4) Generate Code for Real-Time and Advanced Business Case Problems

- **Health Scenario (Medical Diagnosis Agent)**:

```
from langchain_openai import ChatOpenAI
from langchain.agents import AgentExecutor, Tool
from langchain.prompts import PromptTemplate
import requests

class MedicalDiagnosisTool(Tool):
    name = "medical_diagnosis_tool"
    description = "Provides possible diagnosis based on symptoms input"
    arguments = {
        "symptoms": "string",
    }

    def call(self, symptoms):
        response = requests.post("https://api.medicaldiagnosis.com/diagnose",
json={"symptoms": symptoms})
        return response.json()

def diagnose_patient(symptoms):
    tool = MedicalDiagnosisTool()
    agent = AgentExecutor([tool], tools=["medical_diagnosis_tool"], verbose=True)
    result = agent.run(symptoms)
    return result

model = ChatOpenAI(model="gpt-4")
question = "What could be the cause of headache and fever?"

result = diagnose_patient(question)
print(result)
```

- **Retail Scenario (Product Recommendation Agent)**:

```
from langchain.agents import AgentExecutor, Tool
from langchain_openai import ChatOpenAI
import requests

class ProductRecommendationTool(Tool):
    name = "product_recommendation_tool"
    description = "Suggests products based on user preferences"
    arguments = {
        "user_preferences": "string",
    }

    def call(self, user_preferences):
        response =
requests.get(f"https://api.retailstore.com/products?preferences={user_preferences
}")
        return response.json()

def recommend_product(user_preferences):
    tool = ProductRecommendationTool()
    agent = AgentExecutor([tool], tools=["product_recommendation_tool"],
verbose=True)
    result = agent.run(user_preferences)
    return result

model = ChatOpenAI(model="gpt-4")
preferences = "I like casual wear and shoes"
```

```
result = recommend_product(preferences)
print(result)
```

- **Insurance Scenario (Claim Verification Agent)**:

```
from langchain.agents import AgentExecutor, Tool
from langchain_openai import ChatOpenAI
import requests

class InsuranceClaimTool(Tool):
    name = "insurance_claim_tool"
    description = "Verifies insurance claims"
    arguments = {
        "claim_id": "string",
    }

    def call(self, claim_id):
        response = requests.get(f"https://api.insurance.com/claims/{claim_id}")
        return response.json()

def verify_claim(claim_id):
    tool = InsuranceClaimTool()
    agent = AgentExecutor([tool], tools=["insurance_claim_tool"], verbose=True)
    result = agent.run(claim_id)
    return result

model = ChatOpenAI(model="gpt-4")
claim_id = "12345678"

result = verify_claim(claim_id)
print(result)
```

```
from langchain.agents import AgentExecutor, Tool
from langchain_openai import ChatOpenAI
import requests

class HealthDataRetrievalTool(Tool):
    name = "health_data_tool"
    description = "Fetches health-related data based on user query"
    arguments = {
        "user_query": "string",
    }

    def call(self, user_query):
        response =
requests.get(f"https://api.healthdata.com/query?query={user_query}")
        return response.json()

def health_analysis(user_query):
    tool = HealthDataRetrievalTool()
    agent = AgentExecutor([tool], tools=["health_data_tool"], verbose=True)
    result = agent.run(user_query)
    return result

model = ChatOpenAI(model="gpt-4")
query = "What are the symptoms of diabetes?"

result = health_analysis(query)
print(result)
```

Business Case - Health: A health insurance company wants to improve the customer support experience for their policyholders. By integrating a real-time data retrieval system into their customer support portal, they can automatically provide answers to medical queries. For example, if a policyholder asks about symptoms of a disease like diabetes, the system fetches the latest medical data to provide accurate, updated responses without relying on static data, improving both user experience and reducing response time.

```
from langchain.agents import AgentExecutor, Tool
from langchain_openai import ChatOpenAI
import requests

class RetailProductRecommendationTool(Tool):
    name = "retail_product_recommendation_tool"
    description = "Recommends products based on customer preferences"
    arguments = {
        "customer_preferences": "string",
    }
    def call(self, customer_preferences):
        response = 
requests.get(f"https://api.retailstore.com/recommendations?preferences={customer_
preferences}")
        return response.json()

def retail_recommendation(customer_preferences):
    tool = RetailProductRecommendationTool()
    agent = AgentExecutor([tool], tools=["retail_product_recommendation_tool"],
verbose=True)
    result = agent.run(customer_preferences)
    return result

model = ChatOpenAI(model="gpt-4")
preferences = "Looking for winter jackets and boots"

result = retail_recommendation(preferences)
print(result)
```

Business Case - Retail: A large e-commerce retailer wants to personalize the shopping experience for their customers by recommending products based on customer preferences. By using a real-time recommendation system, the platform can suggest relevant items like jackets and boots based on a user’s past purchases and search history. This increases conversion rates, improves customer satisfaction, and boosts sales by giving customers exactly what they’re looking for without them needing to search for it.

```
from langchain.agents import AgentExecutor, Tool
from langchain_openai import ChatOpenAI
import requests

class InsuranceClaimVerificationTool(Tool):
    name = "insurance_claim_verification_tool"
    description = "Verifies insurance claims"
    arguments = {
        "claim_id": "string",
    }
    def call(self, claim_id):
        response = requests.get(f"https://api.insurance.com/claims/{claim_id}")
        return response.json()

def verify_claim(claim_id):
    tool = InsuranceClaimVerificationTool()
    agent = AgentExecutor([tool], tools=["insurance_claim_verification_tool"],
verbose=True)
    result = agent.run(claim_id)
    return result

model = ChatOpenAI(model="gpt-4")
claim_id = "987654321"

result = verify_claim(claim_id)
print(result)
```

Business Case - Insurance: An insurance company wants to streamline the claim verification process to reduce manual workload and increase efficiency. By integrating a real-time claim verification tool, the system can automatically check claim details, validate information, and reduce fraud by cross-checking the claim data with other sources. This minimizes human error and improves claim processing time, resulting in a better customer experience and reducing operational costs.

```
from langchain.agents import AgentExecutor, Tool
from langchain_openai import ChatOpenAI
import requests

class CustomerSupportTool(Tool):
    name = "customer_support_tool"
    description = "Assists with customer inquiries"
    arguments = {
        "customer_query": "string",
    }

    def call(self, customer_query):
        response = requests.post("https://api.customersupport.com/resolve",
json={"query": customer_query})
```

```
        return response.json()

def customer_inquiry_handling(customer_query):
    tool = CustomerSupportTool()
    agent = AgentExecutor([tool], tools=["customer_support_tool"], verbose=True)
    result = agent.run(customer_query)
    return result

model = ChatOpenAI(model="gpt-4")
query = "How can I reset my account password?"

result = customer_inquiry_handling(query)
print(result)
```

Business Case - Customer Support: A company offers an AI-driven customer support chatbot. Customers frequently ask common questions like how to reset their account password. By integrating a real-time API that connects to their backend support system, the bot can instantly provide users with the necessary steps to reset their password without needing to route them to human support agents. This reduces customer frustration and support ticket volumes, leading to improved customer satisfaction and lower operational costs.

```
from langchain.agents import AgentExecutor, Tool
from langchain_openai import ChatOpenAI
import requests

class InventoryManagementTool(Tool):
    name = "inventory_management_tool"
    description = "Manages product stock levels"
    arguments = {
        "product_id": "string",
        "quantity": "integer",
    }

    def call(self, product_id, quantity):
        response = requests.post(f"https://api.inventory.com/update_stock",
json={"product_id": product_id, "quantity": quantity})
        return response.json()
```

```
def manage_inventory(product_id, quantity):
    tool = InventoryManagementTool()
    agent = AgentExecutor([tool], tools=["inventory_management_tool"],
verbose=True)
    result = agent.run({"product_id": product_id, "quantity": quantity})
    return result

model = ChatOpenAI(model="gpt-4")
product_id = "product123"
quantity = 50

result = manage_inventory(product_id, quantity)
print(result)
```

Business Case - Inventory Management: A retail chain wants to improve inventory management by automating stock level updates. Using a real-time inventory management system, the company can automatically update stock levels based on sales data and prevent stockouts. When an item’s inventory runs low, the system can trigger restocking alerts or even automate the restocking process, ensuring smooth operation and preventing lost sales.

```
from langchain.agents import AgentExecutor, Tool
from langchain_openai import ChatOpenAI
import requests

class EmployeeSchedulingTool(Tool):
    name = "employee_scheduling_tool"
    description = "Handles employee schedule management"
    arguments = {
        "employee_id": "string",
        "shift_time": "string",
    }

    def call(self, employee_id, shift_time):
        response = requests.post(f"https://api.employeescheduling.com/schedule",
json={"employee_id": employee_id, "shift_time": shift_time})
        return response.json()

def schedule_employee(employee_id, shift_time):
    tool = EmployeeSchedulingTool()
    agent = AgentExecutor([tool], tools=["employee_scheduling_tool"],
verbose=True)
    result = agent.run({"employee_id": employee_id, "shift_time": shift_time})
    return result

model = ChatOpenAI(model="gpt-4")
employee_id = "emp456"
shift_time = "2024-12-30 09:00:00"

result = schedule_employee(employee_id, shift_time)
print(result)
```

Business Case - Employee Scheduling: A large multinational company needs to manage employee schedules efficiently across multiple departments. By using an AI-powered scheduling tool, HR managers can automatically assign shifts based on employee availability, work preferences, and legal constraints. This reduces scheduling conflicts, ensures operational efficiency, and improves employee satisfaction by reducing manual scheduling errors.

5) Pros and Cons of Using Agents with LLMs and LangChain

Pros:

1. **Automated Decision Making**: Agents can automate complex decision-making processes by evaluating current situations and taking appropriate actions.
2. **Integration with External Resources**: Agents can interact with databases, APIs, and other tools dynamically, enabling real-time responses in business scenarios.
3. **Context-Aware Actions**: Language models embedded in agents allow the agent to make decisions based on context, which is crucial for accurate problem-solving.
4. **Scalability**: Multiple agents can be created to handle different tasks or workflows within a business, such as customer service, product recommendations, or medical diagnostics.
5. **Efficiency**: By automating repetitive tasks, agents save time and resources that would otherwise be spent on manual interventions.

Cons:

1. **Complexity**: Setting up agents requires careful design to ensure the agent makes the right decisions and handles multiple external systems effectively.
2. **Error Propagation**: Agents can make mistakes in choosing actions, and such errors could propagate to the next step in the workflow, causing bigger issues.
3. **Resource Consumption**: Running agents that interact with multiple tools can be resource-intensive, particularly in real-time business cases that require heavy data retrieval and processing.
4. **Maintenance Overhead**: Maintaining agents and tools can be challenging, especially when integrating with external systems that might change or experience downtime.
5. **Uncertainty in Long-Term Outcomes**: Agents rely on models making informed decisions, but they might not always predict long-term consequences accurately, especially in complex business scenarios with evolving data.

---

1. LLMs (Large Language Models)

- **Purpose**: LLMs are the core component for understanding and generating natural language text.
- **Business Goal**: To enable advanced NLP capabilities for a variety of business tasks like chatbots, customer service, content generation, sentiment analysis, and more.
- **Importance**: LLMs are essential for automating complex language-related tasks, improving efficiency, and reducing manual labor, which leads to faster decision-making and lower operational costs.

2. Retrieval Augmented Generation (RAG)

- **Purpose**: Enhances language models by combining them with external knowledge sources to improve response accuracy.
- **Business Goal**: Ensure that business applications, like customer support, are powered by up-to-date, domain-specific information, improving the model's effectiveness and reliability.
- **Importance**: Keeps the model current with real-time data, minimizes hallucinations, and provides tailored, domain-specific insights that lead to better customer service and decision-making.

3. Vector Stores

- **Purpose**: Vector stores efficiently store and retrieve vectorized data (embeddings) for quick search and retrieval.
- **Business Goal**: Make the retrieval process faster and more scalable for large datasets, ensuring the model can quickly access relevant information for tasks like document search, knowledge management, or personalized recommendations.
- **Importance**: Reduces search latency and enhances user experience by enabling quick access to relevant, real-time information.

4. Retrievers

- **Purpose**: Act as intermediaries between the input query and the vector store, selecting the most relevant documents or data points.
- **Business Goal**: To efficiently filter and retrieve relevant data, ensuring the AI-powered systems only process the most useful information, leading to better quality outputs and faster responses.
- **Importance**: Helps in fine-tuning information retrieval processes, improving the accuracy of search results and ensuring that only relevant data is used in further processing.

5. Chains

- **Purpose**: A series of steps that allow different components (e.g., models, retrievers, tools) to work together sequentially in an automated process.
- **Business Goal**: Build end-to-end workflows that integrate various components for tasks like document processing, chatbot responses, or automated report generation.
- **Importance**: Allows businesses to create complex workflows that automate large portions of the decision-making process, improving efficiency and reducing human intervention.

6. Agents

- **Purpose**: Use a language model to choose a sequence of actions to take in a given context.
- **Business Goal**: Enable dynamic and context-aware decision-making for applications such as virtual assistants, automated customer service, and operational automation.
- **Importance**: Agents can simulate decision-making processes, helping businesses scale operations, automate customer service, and optimize resource management.

7. Tools

- **Purpose**: Provide additional functionalities (e.g., APIs, data sources) to augment the capabilities of a language model.
- **Business Goal**: Integrate external systems, data, or services to make the language model more powerful and contextually aware, enhancing tasks like real-time information retrieval, interaction with external systems, or performing actions outside the scope of the LLM.
- **Importance**: Allows businesses to bridge the gap between internal systems (e.g., databases, CRMs) and external knowledge, creating richer, more integrated AI solutions.

8. Memory

- **Purpose**: Enables the system to retain information over multiple interactions with a user, creating a more personalized experience.
- **Business Goal**: Build long-term interactions with customers, providing continuity and better personalization in services such as chatbots, virtual assistants, or personalized recommendation systems.
- **Importance**: Improves customer engagement and satisfaction by remembering user preferences, previous conversations, and context across multiple touchpoints.

9. Prompting

- **Purpose**: The technique of crafting effective input prompts that guide the model to produce the desired outputs.
- **Business Goal**: Ensure that the language model delivers high-quality responses tailored to the business needs, such as answering specific questions, generating creative content, or automating reports.
- **Importance**: Properly crafted prompts improve the efficiency and effectiveness of business applications powered by LLMs, ensuring high-quality outputs with minimal refinement.

10. Human-in-the-Loop (HITL)

- **Purpose**: Involves human oversight and intervention in AI decision-making, especially in cases where the model's response requires validation or refinement.
- **Business Goal**: Enable businesses to maintain control over critical decisions and ensure that AI systems' outputs align with human expertise, ethics, and regulatory requirements.
- **Importance**: Provides an additional layer of quality control, improving the reliability and safety of AI-driven systems, especially in sensitive domains like healthcare, finance, and law.

11. Cost Optimization

- **Purpose**: Strategies to reduce the operational costs associated with using AI systems, including managing compute resources and optimizing model efficiency.
- **Business Goal**: Reduce the overall cost of deploying and maintaining AI systems, making them more accessible and sustainable for businesses of all sizes.
- **Importance**: Optimizing costs ensures that businesses can scale AI models effectively without exceeding budget constraints, making AI adoption more feasible for both large enterprises and startups.

12. Scalability and Performance

- **Purpose**: Ensuring the AI system can handle increasing amounts of data and more complex tasks without degrading performance.
- **Business Goal**: Make sure that as the business grows, the AI systems can scale effectively to handle larger datasets, more customers, and more complex tasks without sacrificing performance.
- **Importance**: Critical for businesses that need AI to scale with growth and increase demand while maintaining efficient, fast, and accurate outputs.

Business Model Integration:

Each of the components mentioned above contributes to key business objectives, including automation, customer engagement, personalization, efficiency, cost reduction, and scalability. The main business goals behind each component include:

- **Improving Operational Efficiency**: Automating repetitive tasks and processes.
- **Enhancing Customer Experience**: Providing fast, personalized, and contextually-aware services.
- **Cost-Effective Growth**: Leveraging AI to scale operations without dramatically increasing costs.
- **Maintaining Data Quality and Relevance**: Ensuring that AI systems use the most up-to-date and relevant information for decision-making.

LangChain's components provide a comprehensive toolkit for building AI-powered applications that drive significant business value across various industries, from healthcare and retail to insurance and finance.
