```python
#  Environment Configuration
from dotenv import load_dotenv
import os
from sqlalchemy.engine.url import make_url # Used to parse and construct database URLs
from langchain_postgres.vectorstores import PGVector # Integration with Postgres + pgvector

# LLM and Core LangChain Tools
from langchain_openai import OpenAIEmbeddings
from langchain_openai import ChatOpenAI

from langchain_core.messages import HumanMessage, SystemMessage
from langchain.load import dumps, loads  # Serialize/deserialize LangChain objects
from langchain_core.prompts import PromptTemplate
from langchain_core.output_parsers import StrOutputParser
from langchain_core.documents import Document # # Standard document format used in LangChai

from typing_extensions import TypedDict # Define structured types for state management
from typing import List  # Specify types for list inputs or outputs
import asyncio # Support asynchronous execution for parallel LLM calls

from langgraph.graph import StateGraph, END # LangGraph tools to define stateful workflows

# Web Search Tool
from langchain_community.tools.tavily_search import TavilySearchResults

# # Import the `trace` decorator from LangSmith to enable tracing of some individual custom
# from langsmith import trace

# Load environment variables from .env file
load_dotenv()

# Access the environment variable
openai_api_key = os.getenv("OPENAI_API_KEY")
connection_string = os.getenv("DB_CONNECTION")
tavily_api_key = os.getenv("TAVILY_API_KEY")

langsmith_api_key = os.getenv("LANGSMITH_API_KEY")

# Enable LangSmith tracing for observability/debugging
os.environ["LANGCHAIN_TRACING"] = "true"
# Set the project name for LangSmith, it will create a new project if it doesn't exist
os.environ["LANGCHAIN_PROJECT"] = "GenAI-Class-Lab8"

# Configure Database Connection
```

```python
# Use the same shared table as from the last lab
shared_connection_string = make_url(connection_string)\
    .set(database="IST345_Drucker_data").render_as_string(hide_password=False) # Leave passu

# Initialize the embedding model
embedding_model = OpenAIEmbeddings(model="text-embedding-3-large")
print("-------- new Conversation ---------")
# Quick check environment variables
if not openai_api_key or not shared_connection_string or not tavily_api_key or not embedding
    print(f"Error: Missing one or more required environment variables") # If so, print out
else:
    print("All environment variables loaded successfully")


# Main LLM for handling complex or creative tasks
llm_gpt = ChatOpenAI(
    model="gpt-4o", # GPT-4o is a powerful model with strong reasoning capabilities
    temperature=0.7,
    api_key=openai_api_key
)

# Lightweight LLM for simple or deterministic tasks
llm_gpt_mini = ChatOpenAI(
    model="gpt-4o-mini",     #  Smaller, faster variant for lightweight tasks
    temperature=0,           # Temperature 0 = fully deterministic output
    api_key=openai_api_key
)

# Connect to the PGVector Vector Store that contains book data.
book_data_vector_store = PGVector(
    embeddings=embedding_model,
    collection_name="Book_data",    # Name of the collection/table in the vector DB
    connection=shared_connection_string, # Use shared DB connection from earlier
    use_jsonb=True,
)

# Define the routing prompt
query_router_prompt_template = PromptTemplate.from_template("""
You are an expert at analyzing user question and deciding which data source is best suited t

1. **Vectorstore**: Use this if the question can be answered by the **existing** content in
   The vectorstore contains information about **{vectorstore_content_summary}**.

---

2. **Websearch**: Use this if the question is **within scope** (see below) but meets **any**
```

```
    - The answer **cannot** be found in the local vectorstore
    - The question requires **more detailed or factual information** than what's available i
    - The topic is **time-sensitive** , **current**, or depends on recent events or updates

---

3. **Chitter-Chatter**: Use this if the question:
   - Is **not related** to the scope below, or
   - Is too **broad, casual, or off-topic** to be answered using vectorstore or websearch.

   Chitter-Chatter is a fallback agent that gives a friendly response and a follow-up to gui

---

Scope Definition:
Relevant questions are those related to **{relevant_scope}**

---

Your Task:
Analyze the user's question. Return a JSON object with one key `"Datasource"` and one value

""")

# Define a summary of what's in the vectorstore
vectorstore_content_summary = """
Peter Drucker's "The Daily Drucker" (2004) provides 366 daily insights and actionable advice
serving as a practical guide for personal and professional growth. "The Effective Executive"
enable executives to achieve effectiveness, focusing on time management, prioritization, and
is a curated collection of Drucker's foundational principles on management, strategy, and le
complex business environments. These books collectively address key aspects of effective man
resources for understanding organizational dynamics and personal productivity.
"""

# Define the topical scope of the system
relevant_scope = """Peter Drucker-related topics, including his management philosophy, leade
and their applications in modern business contexts"""

# Define the multi-query generation prompt
multi_query_generation_prompt = PromptTemplate.from_template("""
You are an AI assistant helping improve document retrieval in a vector-based search system.

---

**Context about the database**
The vectorstore contains the following content:
```

```
{vectorstore_content_summary}

Your goal is to help retrieve **more relevant documents** by rewriting a user's question fro
This helps compensate for the limitations of semantic similarity in vector search.

---

**Instructions**:
Given the original question and the content summary above:
1. Return the **original user question** first.
2. Then generate {num_queries} **alternative versions** of the same question.
    - Rephrase using different word choices, structure, or focus.
    - Use synonyms or shift emphasis slightly, but keep the original meaning.
    - Make sure all rewrites are topically relevant to the database content.

Format requirements:
- Do **not** include bullet points or numbers.
- Each version should appear on a **separate newline**.
- Return **exactly {num_queries} + 1 total questions** (1 original + {num_queries} new ones)

---

**Original user question**: {question}
""")

# Reciprocal Rank Fusion (RRF) Implementation
def reciprocal_rank_fusion(results, k=60):
    fused_scores = {}  # Dictionary to store cumulative RRF scores for each document

    # Iterate through each ranked list of documents
    for docs in results:
        for i, doc in enumerate(docs):
            doc_str = dumps(doc)    # Convert document to a string format (JSON) to use as

            # Initialize the document's fused score if not already present
            if doc_str not in fused_scores:
                fused_scores[doc_str] = 0

            # Apply RRF scoring: 1 / (rank + k), where rank is 1-based
            rank = i + 1  # Adjust rank to start from 1 instead of 0
            fused_scores[doc_str] += 1 / (rank + k)

    # Sort by cumulative RRF score (descending)
    reranked_results = sorted(fused_scores.items(), key=lambda x: x[1], reverse=True)

    # Convert JSON strings back to Document objects and store RRF scores in metadata
```

```
    reranked_documents = []
    for doc_str, score in reranked_results:
        doc = loads(doc_str)  # Convert back to Document object
        doc.metadata["rrf_score"] = score  # Track how the document was ranked
        reranked_documents.append(doc)

    # Return the list of documents with scores embedded in metadata
    return reranked_documents

# Define the relevance grader prompt
relevance_grader_prompt_template = PromptTemplate.from_template("""
You are a a relevance grader evaluating whether a retrieved document is helpful in answering

---

**Retrieved Document**:
{document}

**User Question**:
{question}

---

**Your Task**:
Carefully and  objectively assess whether the document contains any **keyword overlap** or *
Do not require a full answer-just some relevant content is enough to pass.

Return your decision as a JSON object with twith keys: "binary_score".
The "binary_score" should be "pass" or "fail" indicating relevance.
""")

# Define the prompt template for answer generation
answer_generator_prompt_template = PromptTemplate.from_template("""
You are an assistant for question-answering tasks.

---

**Context**:
Use the following information to help answer the question:
{context}

****User Question**:
{question}

---
```

```
**Instructions**:
1. Base your answer primarily on the context provided.
2. If the answer is **not present** in the context, say so explicitly.
3. Keep the answer **concise**, **accurate**, and **focused** on the question.
4. At the end, include a **reference section**:
    - For book-based sources, use **APA-style citations** if possible.
    - For web-based sources, include **page title and URL**.

---

**Answer**:
""")

# Define the hallucination checker prompt
hallucination_checker_prompt_template = PromptTemplate.from_template("""
You are an AI grader evaluating whether a student's answer is factually grounded in the prov

---

**Grading Criteria**:
- **Pass**: The answer is **fully based** on the given FACTS and does not contain any fabric
- **Fail**: The answer information that is **fabricated**, **inaccurate**, or **not supporte

---

**Reference Materials (FACTS)**:
{documents}

**Student's Answer**:
{generation}

---

**Output Instructions**:
Return a JSON object with keys: "binary_score" and "explanation".
- "binary_score": either `"pass"` or `"fail"`
- `"explanation"`: a short justification of the grading decision
""")

# Define the answer verifier prompt
answer_verifier_prompt_template = PromptTemplate.from_template("""
You are an AI grader verifying whether a student's answer correctly addresses the given ques

---

**Grading Criteria**:
```

```
- **Pass**: The answer directly addresses the question, even if it includes additional relev
- **Fail**: The answer is off-topic, misses the point, or does not meaningfully respond to t

---

**question**:
{question}

**Student's Answer**:
{generation}

---

**Output Instructions**:
Return aJSONobject with keys: "binary_score" and "explanation"
- "binary_score": either `"pass"` or `"fail"`
- "explanation":  a short justification for your grading decision
""")

# Define the query rewriter prompt
query_rewriter_prompt_template = PromptTemplate.from_template("""
You are a query optimization expert tasked with rewriting questions to improve vector databa

---

**Context**:
- Original Question: {question}
- Previous Answer (incomplete or unhelpful): {generation}

**Vectorstore Summary**:
{vectorstore_content_summary}

Note: The summary provides context about what's in the database but should not be treated as

---

**Your Task**:
Analyze the original question and the failed answer to identify:
1. What key information the original question was missing
2. Any ambiguities or unclear phrasing
3. Missing context or specialized terminology that should be included
4. Better keywords, phrasing, or terms to improve retrieval

---

**Output Format**:
```

```
Return a JSON object with keys: "rewritten_question" and "explanation".
- "rewritten_question":  A refined version of the user's question optimized for vector searc
- "explanation": A short explanation of how the rewrite improves coverage or clarity
""")

# Define the Chitter-Chatter prompt
chitterchatter_prompt_template = PromptTemplate.from_template("""
You are a friendly assistant designed to keep conversations within the current scope while n

---

**Current Scope**:
{relevant_scope}

Your job is to respond conversationally while gently guiding the user toward relevant and pr

---

**Response Guidelines**:

1. **Casual Chit-Chat**:
  - Respond warmly to greetings and social exchanges.
  - Maintain a natural, friendly tone.

2. **Off-Topic Questions**:
  - Politely acknowledge the question.
  - Mention that it falls outside your current scope.
  - Redirect to a relevant topic or ask a follow-up question within scope.
  - Avoid saying "I don't know" without offering guidance.

3. **In-Scope but Unanswerable Questions**:
  - If the question fits the scope but lacks enough information to answer reliably:
    - Acknowledge the gap.
    - Avoid making unsupported claims.
    - Redirect the user toward a more specific or better-supported question.

---

**Important**:
Never invent or guess answers using general world knowledge.
Your job is to **maintain trust** by keeping the conversation focused and grounded.

Always end with a helpful redirection, question, or suggestion related to the scope above.
""")

# Define the web search tool
```

```python
web_search_tool = TavilySearchResults(
    max_results=5,
    search_depth="advanced",        # Uses advanced search depth for more accurate results
    include_answer=True,            # Include a short answer to original query in the search
    tavily_api_key=tavily_api_key  # You have defined this API key in the .env file.
)

# Define the state of the graph
class GraphState(TypedDict):
    """
    Graph state is a dictionary that contains information we want to propagate to, and modi
    """
    question: str                          # User question
    original_question: str                # Copy of original question
    generation: str                        # LLM generation
    datasource: str                        # Output from router node: Vectorstore, Websearch,
    hallucination_checker_attempts: int  # Number of times hallucination checker has been t
    answer_verifier_attempts: int          # Number of times answer verifier has been trigge
    documents: List[str]                   # List of retrieved documents from vectorstore or
    checker_result: str                    # Result of document relevance check: 'pass' or 'f

# ------------------------ Document Retriever Node ------------------------
def document_retriever(state):
    """
    Retrieves documents relevant to the user's question using multi-query RAG fusion.

    This node performs the following steps:
    - Reformulates the original user question into multiple diverse sub-queries.
    - Executes MMR-based retrieval for each reformulated query.
    - Applies Reciprocal Rank Fusion (RRF) to combine and rerank results.
    - Filters out metadata fields that are internal (like RRF scores).
    - Prepares and returns a list of LangChain `Document` objects to be used in downstream

    Args:
        state (GraphState): The current state of the LangGraph, containing the user's quest

    Returns:
        dict: A dictionary containing a cleaned list of relevant `Document` objects under t
    """
    print("\n---QUERY TRANSLATION AND RAG-FUSION---")

    question = state["question"]
    multi_query_generator = (
    multi_query_generation_prompt   # The prompt defines what the LLM should do
    | llm_gpt                       # An LLM generates query variants
    | StrOutputParser()             # Parses the raw output as a string
```

```
    | (lambda x: x.split("\n"))   # A lambda function to split the result into a list of st
    )
    retrieval_chain_rag_fusion_mmr = (
        multi_query_generator
        | book_data_vector_store.as_retriever(
            search_type="mmr", # Use MMR retrieval to enhance diversity in retrieved documen
            search_kwargs={
                'k': 3,             # Final number of documents to return per query
                'fetch_k': 15,      # Initial candidate pool (larger for better diversity)
                "lambda_mult": 0.5  # Balances relevance (0) and diversity (1)
                }
                ).map()  # Apply MMR retrieval to each reformulated query
        | reciprocal_rank_fusion # Rerank the combined results using RRF
    )

     # Run multi-query RAG + MMR + RRF pipeline to get relevant results
    rag_fusion_mmr_results = retrieval_chain_rag_fusion_mmr.invoke({
        "question": question,
        "num_queries": 3,
        "vectorstore_content_summary": vectorstore_content_summary
        })

     # Display summary of where results came from (for teaching purposes)
    print(f"Total number of results: {len(rag_fusion_mmr_results)}")
    for i, doc in enumerate(rag_fusion_mmr_results, start=1):
        print(f"     Document {i} from `{doc.metadata['source']}`, page {doc.metadata['page

    # Convert retrieved documents into Document objects with metadata and page_content only
    formatted_doc_results = [
        Document(
            metadata={k: v for k, v in doc.metadata.items() if k != 'rrf_score'}, # Remove
            page_content=doc.page_content
        )
        for doc in rag_fusion_mmr_results
    ]

    return {"documents": formatted_doc_results}


# ------------------------ Answer Generator Node ------------------------
def answer_generator(state):
    """
    Generates an answer based on the retrieved documents and user question.

    This node prepares a prompt that includes:
    - The original or rewritten user question
```

```
    - A list of relevant documents (from vectorstore or web search)

    It invokes the main LLM to synthesize a concise and grounded response, returning the res
    for use in later hallucination and usefulness checks.

    Args:
        state (GraphState): The current LangGraph state containing documents and question(s)

    Returns:
        dict: A dictionary with one key `"generation"` containing the LLM-generated answer
    """
    print("\n---ANSWER GENERATION---")

    documents = state["documents"]

    # Use original_question if available (after rewriting), otherwise default to input quest
    original_question = state.get("original_question", 0)
    if original_question != 0:
        question = original_question
    else:
        question = state["question"]

     # Ensure all documents are LangChain Document objects (convert from dicts if needed)
    documents = [
        Document(metadata=doc["metadata"], page_content=doc["page_content"])
        if isinstance(doc, dict) else doc
        for doc in documents
    ]

    # Format the prompt for the answer generator
    answer_generator_prompt = answer_generator_prompt_template.format(
        context=documents,
        question=question
    )

    # Call the LLM to generate the answer
    answer_generation = llm_gpt.invoke(answer_generator_prompt)
    print(f"Answer generation has been generated.")

    return {"generation": answer_generation.content}


# ------------------------ Web Searcher Node ------------------------
def web_search(state):
    """
    Performs a real-time web search and appends results to previously retrieved documents.
```

```
    This node is used when:
    - The vectorstore lacks sufficient relevant information
    - The original question requires current or factual information from the web

    It queries the web using a tool (e.g., Tavily), formats the returned results as LangCha
    `Document` objects, and appends them to the existing document list for downstream answe

    Args:
        state (GraphState): Current graph state with the user's question and optional prior

    Returns:
        dict: Updated state with the combined list of vectorstore and web search documents
    """

    print("\n---WEB SEARCH---")

    question = state["question"]
    documents = state.get("documents", [])

    # Run the web search using the web search tool
    web_results = web_search_tool.invoke(question)

    # Convert raw web search results into a simplified format
    formatted_web_results = [
            {
                "metadata": {
                    "title": result["title"],
                    "url": result["url"]
                },
                "page_content": result["content"]
            }
            for result in web_results
        ]

    # Ensure previous documents are consistently formatted as LangChain Document objects
    documents = [
        Document(metadata=doc["metadata"], page_content=doc["page_content"])
        if isinstance(doc, dict) else doc
        for doc in documents
    ]

    # Append the new web documents
    documents.extend(formatted_web_results)

    print(f"Total number of web search documents: {len(formatted_web_results)}")
```

```
    return {"documents": documents}

# ------------------------ Chitter-Chatter Node ------------------------
def chitter_chatter(state):
    """
    Handles casual, off-topic, or unanswerable in-scope questions using a fallback assistant

    This node is designed to keep the user engaged and politely redirect them toward questio
    that are better suited to the system's capabilities.

    Args:
        state (GraphState): Current graph state containing the user question.

    Returns:
        dict: Response from the Chitter-Chatter agent under the key `"generation"`.
    """
    print("\n---CHIT-CHATTING---")
    question = state["question"]
    chitterchatter_prompt = chitterchatter_prompt_template.format(relevant_scope=relevant_sc
    # Generate a friendly fallback response using the Chitter-Chatter prompt
    chitterchatter_response = llm_gpt_mini.invoke(
        [SystemMessage(chitterchatter_prompt),
         HumanMessage(question)])

    return {"generation": chitterchatter_response.content}


# ------------------------ Adaptive Query Rewrite Node ------------------------
def query_rewriter(state):
    """
    Rewrites the original question if the answer was hallucinated or unhelpful.

    This node helps improve retrieval quality in the second attempt by:
    - Identifying gaps between the original query and the generated answer
    - Generating a clearer, more focused version of the question
    - Keeping a copy of the original for fallback comparison

    Args:
        state (GraphState): Contains the original and current question, and the LLM's previo

    Returns:
        dict: Updated state with the rewritten question and preserved original.
    """
    print("\n---QUERY REWRITE---")

    # Use original question if available, otherwise fall back to input
```

```
    original_question = state.get("original_question", 0)
    if original_question != 0:
        question = original_question
    else:
        question = state["question"]

    generation = state["generation"]

    # Create prompt and invoke the query rewriter
    query_rewriter_prompt = query_rewriter_prompt_template.format(
        question=question,
        generation=generation,
        vectorstore_content_summary=vectorstore_content_summary
    )

    # Use the LLM model to grade the document
    query_rewriter_result = llm_gpt.with_structured_output(method="json_mode").invoke(
        query_rewriter_prompt)

    return {"question": query_rewriter_result['rewritten_question'],
            "original_question": question}


# ------------------------ Retry Counter Node for Hallucination Checker --------------------
def hallucination_checker_tracker(state):
    """
    Tracks how many times the hallucination checker has been triggered.

    This helps avoid infinite loops in the graph by limiting retries after repeated failures

    Args:
        state (GraphState): Current state of the graph including retry metadata.

    Returns:
        dict: Updated state with incremented `hallucination_checker_attempts`.
    """
    num_attempts = state.get("hallucination_checker_attempts", 0)
    return {"hallucination_checker_attempts": num_attempts + 1}

# ------------------------ Retry Counter Node for Answer Verifier ------------------------
def answer_verifier_tracker(state):
    """
    Tracks how many times the answer usefulness checker has been triggered.

    This node helps the workflow know when to stop trying to rewrite queries
    after repeated failures to generate an appropriate answer.
```

```
    Args:
        state (GraphState): Current state including verification metadata.

    Returns:
        dict: Updated state with incremented `answer_verifier_attempts`.
    """
    num_attempts = state.get("answer_verifier_attempts", 0)
    return {"answer_verifier_attempts": num_attempts + 1}

# ------------------------ Routing Decision  ------------------------
def route_question(state):
    """
    Routes the user question to the appropriate agent based on the Query Router's classifica

    Args:
        state (GraphState): Contains the user's input question.

    Returns:
        str: One of 'Vectorstore', 'Websearch', or 'Chitter-Chatter'.
    """
    print("---ROUTING QUESTION---")
    question = state["question"]
    query_router_prompt = query_router_prompt_template.format(
    relevant_scope=relevant_scope,
    vectorstore_content_summary=vectorstore_content_summary)
    route_question_response = llm_gpt.with_structured_output(method="json_mode").invoke(
        [SystemMessage(query_router_prompt),
        HumanMessage(question)]
    )

    parsed_router_output = route_question_response["Datasource"]

    if parsed_router_output == "Websearch":
        print("---ROUTING QUESTION TO WEB SEARCH---")
        return "Websearch"
    elif parsed_router_output == "Vectorstore":
        print("---ROUTING QUESTION TO VECTORSTORE---")
        return "Vectorstore"
    elif parsed_router_output == "Chitter-Chatter":
        print("---ROUTING QUESTION TO CHITTER-CHATTER---")
        return "Chitter-Chatter"


# ------------------------ Async document relevance grading  ------------------------
async def grade_documents_parallel(state):
```

```
    """
    Grades retrieved documents asynchronously to determine their relevance to the user's qu

    Documents are processed in parallel using async calls. If 50% or more are irrelevant,
    the system flags this as a failure, triggering a web search in the next step.

    Args:
        state (GraphState): Contains the documents and question.

    Returns:
        dict: Updated state with filtered documents and a `"checker_result"` of 'pass' or '
    """
    print("---CHECK DOCUMENT RELEVANCE TO QUESTION---")
    question = state["question"]
    documents = state["documents"]

     # Inner coroutine that grades one document at a time using the relevance grader prompt
    async def grade_document(doc, question):
        relevance_grader_prompt = relevance_grader_prompt_template.format(
            document=doc,
            question=question
        )
        grader_result = await llm_gpt_mini.with_structured_output(method="json_mode").ainvok
            relevance_grader_prompt)
        return grader_result

    # Create async tasks for grading all documents
    tasks = [grade_document(doc, question) for doc in documents]

    # Run all tasks concurrently
    results = await asyncio.gather(*tasks)

    filtered_docs = []

   # Collect only documents marked as "pass"
    for i, score in enumerate(results):
        if score["binary_score"].lower() == "pass":
            print(f"---GRADE: DOCUMENT RELEVANT--- {score['binary_score']}")
            filtered_docs.append(documents[i]) # only keep the relevant ones
        else:
            print("---GRADE: DOCUMENT NOT RELEVANT---")

    # Analyze how many documents were filtered out
    total_docs = len(documents)
    relevant_docs = len(filtered_docs)
```

```
    if total_docs > 0:
        filtered_out_percentage = (total_docs - relevant_docs) / total_docs

        # If more than 50% of documents were irrelevant, fail and fall back to web search
        checker_result = "fail" if filtered_out_percentage >= 0.5 else "pass"
        print(f"---FILTERED OUT {filtered_out_percentage*100:.1f}% OF IRRELEVANT DOCUMENTS--
        print(f"---**{checker_result}**---")
    else:
        # If no documents were retrieved at all, treat as automatic failure
        checker_result = "fail"
        print("---NO DOCUMENTS AVAILABLE, WEB SEARCH TRIGGERED---")

    return {"documents": filtered_docs, "checker_result": checker_result}


# ------------------------ Decide whether to generate or fallback ------------------------
def decide_to_generate(state):
    """
    Conditional edge function used after document relevance grading.

    It checks the `checker_result` from the previous step:
    - If the result is 'fail' (indicating that a majority of documents were irrelevant),
      it triggers a fallback to web search for more reliable context.
    - If the result is 'pass', it proceeds to the answer generation node.

    Args:
        state (GraphState): Includes the 'checker_result' from the document grading step.

    Returns:
        str: Either 'generate' or 'Websearch', used to transition to the next node in the L
    """
    print("---CHECK GENERATION CONDITION---")
    checker_result = state["checker_result"]

    if checker_result == "fail":
        print(
            "---DECISION: MORE THAN HALF OF THE DOCUMENTS ARE IRRELEVANT TO QUESTION, NOW IN
        )
        return "Websearch"
    else:
        # We have relevant documents, so generate answer
        print("---DECISION: GENERATE---")
        return "generate"


# # ------------------------ Final Answer Validation ------------------------
```

```python
def check_generation_vs_documents_and_question(state):
    """
    Conditional edge function verifies the quality of the generated answer against two crite
    - Grounded in the retrieved documents (hallucination check)
    - Relevant to the original user question (answer verifier)

    Depending on the result, this function controls whether the system proceeds, rewrites th
    retries answer generation, or stops after exceeding retry limits.

    Args:
        state (GraphState): Includes question, generated answer, documents, and retry counte

    Returns:
        str: One of the route labels used in LangGraph transitions:
            - 'useful': Answer is grounded and relevant
            - 'not useful': Answer is grounded but does not address the question
            - 'not supported': Answer is not grounded (hallucination)
            - 'max retries': Too many failed attempts, abort or fallback
    """

    print("---CHECK HALLUCINATIONS WITH DOCUMENTS---")

    # Use original rewritten question if present; otherwise use latest version
    question = state["question"]
    original_question = state.get("original_question", 0)

    if original_question != 0:
        question = original_question
    else:
        question = state["question"]

    documents = state["documents"]
    generation = state["generation"]

    # Retry counters
    hallucination_checker_attempts = state.get("hallucination_checker_attempts", 0)
    answer_verifier_attempts = state.get("answer_verifier_attempts", 0)

    # Run hallucination checker: does the answer come from the documents?
    hallucination_checker_prompt = hallucination_checker_prompt_template.format(
        documents=documents,
        generation=generation
    )
    hallucination_checker_result = llm_gpt_mini.with_structured_output(method="json_mode").i
    hallucination_checker_prompt)
```

```
    # Helper to format "1st", "2nd", etc.
    def ordinal(n):
        return f"{n}{'th' if 10 <= n % 100 <= 20 else {1:'st', 2:'nd', 3:'rd'}.get(n % 10, '

    # If generation is grounded (pass hallucination check)
    if hallucination_checker_result['binary_score'].lower() == "pass":
        print("---DECISION: GENERATION IS GROUNDED IN DOCUMENTS---")

        # Now check if it answers the question usefully
        print("---VERIFY ANSWER WITH QUESTION---")
        # Test using question and generation from above
        answer_verifier_prompt = answer_verifier_prompt_template.format(
            question=question,
            generation=generation
        )
        answer_verifier_result = llm_gpt_mini.with_structured_output(method="json_mode").inv
        answer_verifier_prompt)

        # If answer is grounded AND relevant, return final result
        if answer_verifier_result['binary_score'].lower() == "pass":
            print("---DECISION: GENERATION ADDRESSES QUESTION---")
            return "useful"

        # If max attempts reached for usefulness check, exit
        elif answer_verifier_attempts > 1:
            print("---DECISION: MAX RETRIES REACHED---")
            return "max retries"

        # Otherwise, try query rewrite and retry generation
        else:
            print("---DECISION: GENERATION DOES NOT ADDRESS QUESTION, RE-WRITE QUERY---")
            print(f"This is the {ordinal(answer_verifier_attempts+1)} attempt.")
            return "not useful"

    # If generation is NOT grounded and retry limit exceeded
    elif hallucination_checker_attempts > 1:
        print("---DECISION: MAX RETRIES REACHED---")
        return "max retries"

    # If answer is not grounded but we can still retry
    else:
        print("---DECISION: GENERATION IS NOT GROUNDED IN DOCUMENTS, RE-TRY---")
        print(f"This is the {ordinal(hallucination_checker_attempts+1)} attempt.")
        return "not supported"

# Initialize the graph with shared state structure
```

```
workflow = StateGraph(GraphState)

# === Add agent nodes ===
workflow.add_node("WebSearcher", web_search)                    # web search
workflow.add_node("DocumentRetriever", document_retriever)      # Multi-query RAG + MMR + Rl
workflow.add_node("RelevanceGrader", grade_documents_parallel)  # Async document evaluation
workflow.add_node("AnswerGenerator", answer_generator)          # Generate grounded response
workflow.add_node("QueryRewriter", query_rewriter)              # Rewrite query if generatio
workflow.add_node("ChitterChatter", chitter_chatter)            #  Fallback for unsupported

# === Add retry tracker nodes ===
workflow.add_node("HallucinationCheckerFailed", hallucination_checker_tracker)
workflow.add_node("AnswerVerifierFailed", answer_verifier_tracker)

# === Entry point: Route query to appropriate agent ===
workflow.set_conditional_entry_point(
    route_question,
    {
        "Websearch": "WebSearcher",
        "Vectorstore": "DocumentRetriever",
        "Chitter-Chatter": "ChitterChatter",
    },
)

# === Node transitions ===
workflow.add_edge("DocumentRetriever", "RelevanceGrader")           # Retrieve → Grade
workflow.add_edge("WebSearcher", "AnswerGenerator")                 # Web search → Generate

workflow.add_edge("HallucinationCheckerFailed", "AnswerGenerator")  # Retry after failed gr
workflow.add_edge("AnswerVerifierFailed", "QueryRewriter")          # Retry after poor answe
workflow.add_edge("QueryRewriter", "DocumentRetriever")             # Rewritten query → new
workflow.add_edge("ChitterChatter", END)                            # End if fallback agent

# === Conditional routing after document grading ===
workflow.add_conditional_edges(
    "RelevanceGrader",
    decide_to_generate,
    {
        "Websearch": "WebSearcher",         # Too many irrelevant docs → Web search
        "generate": "AnswerGenerator",      # Good enough → Proceed to generate
    },
)

# === Conditional routing after generation quality checks ===
workflow.add_conditional_edges(
    "AnswerGenerator",
```

```
    check_generation_vs_documents_and_question,
    {
        "not supported": "HallucinationCheckerFailed",  # Hallucinated → Retry generation
        "useful": END,                                   # Success
        "not useful": "AnswerVerifierFailed",            # Off-topic → Rewrite & retry
        "max retries": "ChitterChatter"                  # Stop after too many failures
    },
)

# --- Compile the graph ---
def get_workflow():
    return workflow
```