Assignment 2: Advanced RAG Techniques

Day 6 Session 2 - Advanced RAG Fundamentals

OBJECTIVE: Implement advanced RAG techniques including postprocessors, response synthesizers, and structured outputs.

LEARNING GOALS:

- Understand and implement node postprocessors for filtering and reranking
- Learn different response synthesis strategies (TreeSummarize, Refine)
- Create structured outputs using Pydantic models
- Build advanced retrieval pipelines with multiple processing stages

DATASET: Use the same data folder as Assignment 1 (`Day_6/session_2/data/`)

PREREQUISITES: Complete Assignment 1 first

INSTRUCTIONS:

1. Complete each function by replacing the TODO comments with actual implementation
2. Run each cell after completing the function to test it
3. The answers can be found in the `03_advanced_rag_techniques.ipynb` notebook
4. Each technique builds on the previous one

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

```
%pip install -q -r '/content/drive/MyDrive/RAGDemo/requirements.txt'
```

```
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 803.2/803.2 kB 13.0 MB/s eta 0:00:00
  Installing build dependencies ... done
  Getting requirements to build wheel ... done
  Preparing metadata (pyproject.toml) ... done
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 182.0/182.0 kB 6.5 MB/s eta 0:00:00
WARNING: huggingface-hub 1.3.7 does not provide the extra 'inference'
    ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 47.2/47.2 MB 18.1 MB/s eta 0:00:00
    ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 3.3/3.3 MB 72.8 MB/s eta 0:00:00
    ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1.6/1.6 MB 60.2 MB/s eta 0:00:00
    ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 278.0/278.0 kB 19.7 MB/s eta 0:00:00
    ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 11.9/11.9 MB 86.1 MB/s eta 0:00:00
```

```
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 303.3/303.3 kB 13.5 MB/s eta 0:00:00
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 51.8/51.8 kB 2.2 MB/s eta 0:00:00
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 188.3/188.3 MB 6.0 MB/s eta 0:00:00
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 57.6/57.6 MB 9.5 MB/s eta 0:00:00
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 4.2/4.2 MB 23.1 MB/s eta 0:00:00
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 96.8/96.8 kB 7.4 MB/s eta 0:00:00
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 63.9/63.9 kB 4.1 MB/s eta 0:00:00
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 329.1/329.1 kB 12.4 MB/s eta 0:00:00
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 88.0/88.0 kB 4.6 MB/s eta 0:00:00
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 51.0/51.0 kB 2.3 MB/s eta 0:00:00
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 150.7/150.7 kB 5.9 MB/s eta 0:00:00
  Building wheel for openai-whisper (pyproject.toml) ... done
```

```python
from google.colab import userdata
import os

os.environ["OPENROUTER_API_KEY"] = userdata.get("OPENROUTER_API_KEY")

print("✅ Initialized Environment successfully!")
```

```
✅ Initialized Environment successfully!
```

```python
# Import required libraries for advanced RAG
import os
from pathlib import Path
from typing import Dict, List, Optional, Any
from pydantic import BaseModel, Field

# Core LlamaIndex components
from llama_index.core import SimpleDirectoryReader, VectorStoreIndex, StorageCc
from llama_index.core.query_engine import RetrieverQueryEngine
from llama_index.core.retrievers import VectorIndexRetriever

# Vector store
from llama_index.vector_stores.lancedb import LanceDBVectorStore

# Embeddings and LLM
from llama_index.embeddings.huggingface import HuggingFaceEmbedding
from llama_index.llms.openrouter import OpenRouter

# Advanced RAG components (we'll use these in the assignments)
from llama_index.core.postprocessor import SimilarityPostprocessor
from llama_index.core.response_synthesizers import TreeSummarize, Refine, Compa
from llama_index.core.output_parsers import PydanticOutputParser

print("✅ Advanced RAG libraries imported successfully!")
```

```
✅ Advanced RAG libraries imported successfully!
```

```
# Configure Advanced RAG Settings (Using OpenRouter)
def setup_advanced_rag_settings():
    """
    Configure LlamaIndex with optimized settings for advanced RAG.
    Uses local embeddings and OpenRouter for LLM operations.
    """
    # Check for OpenRouter API key
    api_key = os.getenv("OPENROUTER_API_KEY")
    if not api_key:
        print("⚠️  OPENROUTER_API_KEY not found - LLM operations will be limit
        print("   You can still complete postprocessor and retrieval exercises"
    else:
        print("✅ OPENROUTER_API_KEY found - full advanced RAG functionality a

        # Configure OpenRouter LLM
        Settings.llm = OpenRouter(
            api_key=api_key,
            model="gpt-4o",
            temperature=0.3  # Lower temperature for more consistent responses
        )

    # Configure local embeddings (no API key required)
    Settings.embed_model = HuggingFaceEmbedding(
        model_name="BAAI/bge-small-en-v1.5",
        trust_remote_code=True
    )

    # Advanced RAG configuration
    Settings.chunk_size = 512  # Smaller chunks for better precision
    Settings.chunk_overlap = 50

    print("✅ Advanced RAG settings configured")
    print("   - Chunk size: 512 (optimized for precision)")
    print("   - Using local embeddings for cost efficiency")
    print("   - OpenRouter LLM ready for response synthesis")

# Setup the configuration
setup_advanced_rag_settings()
```

```
✅ OPENROUTER_API_KEY found - full advanced RAG functionality available
modules.json: 100%  349/349 [00:00<00:00, 26.7kB/s]
config_sentence_transformers.json: 100%  124/124 [00:00<00:00, 12.7kB/s]
README.md:  94.8k/? [00:00<00:00, 6.06MB/s]
sentence_bert_config.json: 100%  52.0/52.0 [00:00<00:00, 4.85kB/s]
config.json: 100%  743/743 [00:00<00:00, 57.6kB/s]
model.safetensors: 100%  133M/133M [00:01<00:00, 173MB/s]
Loading weights: 100%  199/199 [00:00<00:00, 528.62it/s, Materializing param=pooler.dense.weight]
BertModel LOAD REPORT from: BAAI/bge-small-en-v1.5
Key                     | Status     |  |
------------------------+------------+--+-
embeddings.position_ids | UNEXPECTED |  |

Notes:
- UNEXPECTED    :can be ignored when loading from different task/architecture; r
tokenizer_config.json: 100%  366/366 [00:00<00:00, 13.4kB/s]
vocab.txt:  232k/? [00:00<00:00, 5.39MB/s]
tokenizer.json:  711k/? [00:00<00:00, 30.6MB/s]
special_tokens_map.json: 100%  125/125 [00:00<00:00, 11.7kB/s]
config.json: 100%  190/190 [00:00<00:00, 18.0kB/s]
✅ Advanced RAG settings configured
   - Chunk size: 512 (optimized for precision)
   - Using local embeddings for cost efficiency
   - OpenRouter LLM ready for response synthesis
```

```python
# Setup: Create index from Assignment 1 (reuse the basic functionality)
def setup_basic_index(data_folder: str = "/content/drive/MyDrive/RAGDemo/papers
    """
    Create a basic vector index that we'll enhance with advanced techniques.
    This reuses the concepts from Assignment 1.
    """
    # Create vector store
    vector_store = LanceDBVectorStore(
        uri="./advanced_rag_vectordb",
        table_name="documents"
    )

    # Load documents
    if not Path(data_folder).exists():
        print(f"❌ Data folder not found: {data_folder}")
        return None

    reader = SimpleDirectoryReader(input_dir=data_folder, recursive=True)
```

```
        documents = reader.load_data()

        # Create storage context and index
        storage_context = StorageContext.from_defaults(vector_store=vector_store)
        index = VectorStoreIndex.from_documents(
            documents,
            storage_context=storage_context,
            show_progress=True
        )

        print(f"✅ Basic index created with {len(documents)} documents")
        print("   Ready for advanced RAG techniques!")
        return index

    # Create the basic index
    print("📁 Setting up basic index for advanced RAG...")
    index = setup_basic_index()

    if index:
        print("🚀 Ready to implement advanced RAG techniques!")
    else:
        print("❌ Failed to create index - check data folder path")
```

```
WARNING:llama_index.vector_stores.lancedb.base:Table documents doesn't exist yet
📁 Setting up basic index for advanced RAG...
Parsing nodes: 100%                                  229/229 [00:03<00:00, 172.79it/s]
Generating embeddings: 100%                          733/733 [06:53<00:00,  1.90it/s]
✅ Basic index created with 229 documents
   Ready for advanced RAG techniques!
🚀 Ready to implement advanced RAG techniques!
```

1. Node Postprocessors - Similarity Filtering

Concept: Postprocessors refine retrieval results after the initial vector search. The `SimilarityPostprocessor` filters out chunks that fall below a relevance threshold.

Why it matters: Raw vector search often returns some irrelevant results. Filtering improves precision and response quality.

Complete the function below to create a query engine with similarity filtering.

```
    def create_query_engine_with_similarity_filter(index, similarity_cutoff: float
        """
        Create a query engine that filters results based on similarity scores.

        TODO: Complete this function to create a query engine with similarity postp
```

```
        HINT: Use index.as_query_engine() with node_postprocessors parameter contai

        Args:
            index: Vector index to query
            similarity_cutoff: Minimum similarity score (0.0 to 1.0)
            top_k: Number of initial results to retrieve before filtering

        Returns:
            Query engine with similarity filtering
        """
        if not index:
            print("No index provided for query engine")
            return None

        try:
            # Create similarity postprocessor with the cutoff threshold
            similarity_processor = SimilarityPostprocessor(similarity_cutoff=simila

            # Create query engine with similarity filtering
            # Retrieve more documents initially (top_k) to account for filtering
            query_engine = index.as_query_engine(
                similarity_top_k=top_k,
                node_postprocessors=[similarity_processor]
            )

            print(f"✓ Query engine with similarity filtering created")
            print(f"  - Similarity cutoff: {similarity_cutoff}")
            print(f"  - Initial retrieval (top_k): {top_k}")

            return query_engine

        except Exception as e:
            print(f"Error creating query engine with similarity filter: {e}")
            return None

    # Test the function
    if index:
        filtered_engine = create_query_engine_with_similarity_filter(index, similar

        if filtered_engine:
            print("✅ Query engine with similarity filtering created")

            # Test query
            test_query = "What are the benefits of AI agents?"
            print(f"\n🔍 Testing query: '{test_query}'")

            # Uncomment when implemented:
            # response = filtered_engine.query(test_query)
            # print(f"📝 Response: {response}")
            print("   (Complete the function above to test the response)")
        else:
```

```
        print("❌ Failed to create filtered query engine")
else:
    print("❌ No index available - run previous cells first")
```

```
✓ Query engine with similarity filtering created
  - Similarity cutoff: 0.3
  - Initial retrieval (top_k): 10
✅ Query engine with similarity filtering created

🔍 Testing query: 'What are the benefits of AI agents?'
   (Complete the function above to test the response)
```

2. Response Synthesizers - TreeSummarize

Concept: Response synthesizers control how retrieved information becomes final answers. `TreeSummarize` builds responses hierarchically, ideal for complex analytical questions.

Why it matters: Different synthesis strategies work better for different query types. TreeSummarize excels at comprehensive analysis and long-form responses.

Complete the function below to create a query engine with TreeSummarize response synthesis.

```
def create_query_engine_with_tree_summarize(index, top_k: int = 5):
    """
    Create a query engine that uses TreeSummarize for comprehensive responses.

    TODO: Complete this function to create a query engine with TreeSummarize sy
    HINT: Create a TreeSummarize instance, then use index.as_query_engine() wit

    Args:
        index: Vector index to query
        top_k: Number of results to retrieve

    Returns:
        Query engine with TreeSummarize synthesis
    """
    if not index:
        print("No index provided for query engine")
        return None

    try:
        # TO DO: Create TreeSummarize response synthesizer
        tree_synthesizer = TreeSummarize(verbose=True)

        # To DO: Create query engine with the TreeSummarize synthesizer
```

```
            query_engine = index.as_query_engine(
                similarity_top_k=top_k,
                response_synthesizer=tree_synthesizer
            )

            print(f"✓ Query engine with TreeSummarize synthesis created")
            print(f"  - Response synthesizer: TreeSummarize")
            print(f"  - Retrieval count (top_k): {top_k}")
            print(f"  - Ideal for: Comprehensive analysis and long-form responses")

            return query_engine

        except Exception as e:
            print(f"Error creating query engine with TreeSummarize: {e}")
            return None

    # Test the function
    if index:
        tree_engine = create_query_engine_with_tree_summarize(index)

        if tree_engine:
            print("✅ Query engine with TreeSummarize created")

            # Test with a complex analytical query
            analytical_query = "Compare the advantages and disadvantages of differe
            print(f"\n🔍 Testing analytical query: '{analytical_query}'")

            # Uncomment when implemented:
            # response = tree_engine.query(analytical_query)
            # print(f"📝 TreeSummarize Response:\n{response}")
            print("   (Complete the function above to test comprehensive analysis)"
        else:
            print("❌ Failed to create TreeSummarize query engine")
    else:
        print("❌ No index available - run previous cells first")
```

```
✓ Query engine with TreeSummarize synthesis created
  - Response synthesizer: TreeSummarize
  - Retrieval count (top_k): 5
  - Ideal for: Comprehensive analysis and long-form responses
✅ Query engine with TreeSummarize created

🔍 Testing analytical query: 'Compare the advantages and disadvantages of diffe
   (Complete the function above to test comprehensive analysis)
```

3. Structured Outputs with Pydantic Models

Concept: Structured outputs ensure predictable, parseable responses using Pydantic models. This is essential for API endpoints and data pipelines.

Why it matters: Instead of free-text responses, you get type-safe, validated data structures that applications can reliably process.

Complete the function below to create a structured output system for extracting research paper information.

```
from pydantic import BaseModel, Field
from typing import List

# First, define the Pydantic models for structured outputs
class ResearchPaperInfo(BaseModel):
    """Structured information about a research paper or AI concept."""
    title: str = Field(description="The main title or concept name")
    key_points: List[str] = Field(description="3-5 main points or findings")
    applications: List[str] = Field(description="Practical applications or use
    summary: str = Field(description="Brief 2-3 sentence summary")

# Import the missing component
from llama_index.core.program import LLMTextCompletionProgram

def create_structured_output_program(output_model: BaseModel = ResearchPaperInf
    """
    Create a structured output program using Pydantic models.

    TODO: Complete this function to create a structured output program.
    HINT: Use LLMTextCompletionProgram.from_defaults() with PydanticOutputParse

    Args:
        output_model: Pydantic model class for structured output

    Returns:
        LLMTextCompletionProgram that returns structured data
    """
    if not Settings.llm:
        print("⚠️  LLM not configured - cannot create structured output progra
        print("   Please set OPENROUTER_API_KEY to use this feature")
        return None

    try:
        # Create output parser with the Pydantic model
        output_parser = PydanticOutputParser(output_model)

        # Define prompt template for structured extraction
        prompt_template = f"""
Based on the provided context, extract and structure the information about the

Context:
```

```
{{context}}

Query:
{{query}}

Please extract the following information and format it as requested:
- title: The main title or concept name
- key_points: 3-5 main points or findings (as a list)
- applications: Practical applications or use cases (as a list)
- summary: A brief 2-3 sentence summary

{output_parser.get_format_string()}
"""

        # Create the structured output program
        program = LLMTextCompletionProgram.from_defaults(
            output_parser=output_parser,
            prompt_template_str=prompt_template,
            llm=Settings.llm,
            verbose=True
        )

        print(f"✓ Structured output program created with {output_model.__name_
        print(f"  - Output model: {output_model.__name__}")
        print(f"  - Fields: {list(output_model.model_fields.keys())}")

        return program

    except Exception as e:
        print(f"Error creating structured output program: {e}")
        return None

# Test the function
if index:
    structured_program = create_structured_output_program(ResearchPaperInfo)

    if structured_program:
        print("✅ Structured output program created")

        # Test with retrieval and structured extraction
        structure_query = "Tell me about AI agents and their capabilities"
        print(f"\n🔍 Testing structured query: '{structure_query}'")

        # Get context for structured extraction (Uncomment when implemented)
        retriever = VectorIndexRetriever(index=index, similarity_top_k=3)
        nodes = retriever.retrieve(structure_query)
        context = "\n".join([node.text for node in nodes])

        # Uncomment when implemented:
        response = structured_program(context=context, query=structure_query)
        print(f"📊 Structured Response:\n{response}")
```

```
            print("   (Complete the function above to get structured JSON output)")

            print("\n💡 Expected output format:")
            print("   - title: String")
            print("   - key_points: List of strings")
            print("   - applications: List of strings")
            print("   - summary: String")
        else:
            print("❌ Failed to create structured output program")
    else:
        print("❌ No index available - run previous cells first")
```

```
✓ Structured output program created with ResearchPaperInfo
  - Output model: ResearchPaperInfo
  - Fields: ['title', 'key_points', 'applications', 'summary']
✅ Structured output program created

🔍 Testing structured query: 'Tell me about AI agents and their capabilities'
📊 Structured Response:
title='AI Agents and Agentic AI: Capabilities and Challenges' key_points=['AI Ag
   (Complete the function above to get structured JSON output)

💡 Expected output format:
   - title: String
   - key_points: List of strings
   - applications: List of strings
   - summary: String
```

```
response
```

```
ResearchPaperInfo(title='AI Agents and Agentic AI: Capabilities and
Challenges', key_points=['AI Agents are used in knowledge retrieval, email
automation, and report summarization.', 'Agentic AI is applied in research
assistance, robotic swarms, and strategic business planning.', 'Challenges for
AI Agents include hallucination, prompt brittleness, and limited planning
ability.', 'Agentic AI faces risks such as misuse, unemployment, and threats to
human well-being.', 'There are open problems like scaling up agent counts and
the potential path to AGI.'], applications=['Knowledge retrieval', 'Email
automation', 'Report summarization', 'Research assistants', 'Robotic swarms',
'Strategic business planning'], summary='The paper explores the deployment of
AI Agents and Agentic AI across various domains, highlighting their
applications and inherent challenges. AI Agents are primarily used for tasks
like knowledge retrieval and automation, while Agentic AI is applied in more
complex scenarios such as research assistance and strategic planning. The paper
also addresses the limitations and risks associated with these technologies,
including issues of scalability and potential societal impacts.')
```

4. Advanced Pipeline - Combining All Techniques

Concept: Combine multiple advanced techniques into a single powerful query engine: similarity filtering + response synthesis + structured output.

Why it matters: Production RAG systems often need multiple techniques working together for optimal results.

Complete the function below to create a comprehensive advanced RAG pipeline.

```
def create_advanced_rag_pipeline(index, similarity_cutoff: float = 0.3, top_k:
    """
    Create a comprehensive advanced RAG pipeline combining multiple techniques.

    TODO: Complete this function to create the ultimate advanced RAG query engi
    HINT: Combine SimilarityPostprocessor + TreeSummarize using index.as_query_

    Args:
        index: Vector index to query
        similarity_cutoff: Minimum similarity score for filtering
        top_k: Number of initial results to retrieve

    Returns:
        Advanced query engine with filtering and synthesis combined
    """
    if not index:
        print("No index provided for advanced pipeline")
        return None

    try:
        # TODO: Create similarity postprocessor to remove low-similarity nodes
        similarity_processor = SimilarityPostprocessor(similarity_cutoff=simila

        # TODO: Create TreeSummarize for hierarchical response synthesis
        tree_synthesizer = TreeSummarize(verbose=True)

        # TODO: Create the comprehensive query engine combining both techniques
        advanced_engine = index.as_query_engine(
            similarity_top_k=top_k,
            node_postprocessors=[similarity_processor],
            response_synthesizer=tree_synthesizer
        )

        print(f"✓ Advanced RAG pipeline created")
        print(f"  - Similarity cutoff: {similarity_cutoff}")
        print(f"  - Initial retrieval (top_k): {top_k}")
        print(f"  - Synthesizer: TreeSummarize")

        return advanced_engine
    except Exception as e:
        print(f"Error creating advanced RAG pipeline: {e}")
        return None
```

```
# Test the comprehensive pipeline
if index:
    advanced_pipeline = create_advanced_rag_pipeline(index)

    if advanced_pipeline:
        print("✅ Advanced RAG pipeline created successfully!")
        print("   🔧 Similarity filtering: ✅")
        print("   🌳 TreeSummarize synthesis: ✅")

        # Test with complex query
        complex_query = "Analyze the current state and future potential of AI a
        print(f"\n🔍 Testing complex query: '{complex_query}'")

        # Uncomment when implemented:
        response = advanced_pipeline.query(complex_query)
        print(f"🚀 Advanced RAG Response:\n{response}")
        print("   (Complete the function above to test the full pipeline)")

        print("\n🎯 This should provide:")
        print("   - Filtered relevant results only")
        print("   - Comprehensive analytical response")
        print("   - Combined postprocessing and synthesis")
    else:
        print("❌ Failed to create advanced RAG pipeline")
else:
    print("❌ No index available - run previous cells first")
```

```
✓ Advanced RAG pipeline created
  - Similarity cutoff: 0.3
  - Initial retrieval (top_k): 10
  - Synthesizer: TreeSummarize
✅ Advanced RAG pipeline created successfully!
   🔧 Similarity filtering: ✅
   🌳 TreeSummarize synthesis: ✅

🔍 Testing complex query: 'Analyze the current state and future potential of AI
2 text chunks after repacking
1 text chunks after repacking
🚀 Advanced RAG Response:
AI agent technologies are currently in a promising state, characterized by advan

The future potential of AI agent technologies is significant, with a focus on ov

Future research is expected to optimize agent workflows, enhance memory mechanis
   (Complete the function above to test the full pipeline)

🎯 This should provide:
   - Filtered relevant results only
   - Comprehensive analytical response
   - Combined postprocessing and synthesis
```

5. Final Test - Compare Basic vs Advanced RAG

Once you've completed all the functions above, run this cell to compare basic RAG with your advanced techniques.

```
# Final comparison: Basic vs Advanced RAG
print("🚀 Advanced RAG Techniques Assignment - Final Test")
print("=" * 60)

# Test queries for comparison
test_queries = [
    "What are the key capabilities of AI agents?",
    "How do you evaluate agent performance metrics?",
    "Explain the benefits and challenges of multimodal AI systems"
]

# Check if all components were created
components_status = {
    "Basic Index": index is not None,
    "Similarity Filter": 'filtered_engine' in locals() and filtered_engine is r
    "TreeSummarize": 'tree_engine' in locals() and tree_engine is not None,
    "Structured Output": 'structured_program' in locals() and structured_progra
    "Advanced Pipeline": 'advanced_pipeline' in locals() and advanced_pipeline
}

print("\n📊 Component Status:")
for component, status in components_status.items():
    status_icon = "✅" if status else "❌"
    print(f"   {status_icon} {component}")

# Create basic query engine for comparison
if index:
    print("\n🔍 Creating basic query engine for comparison...")
    basic_engine = index.as_query_engine(similarity_top_k=5)

    print("\n" + "=" * 60)
    print("🆚 COMPARISON: Basic vs Advanced RAG")
    print("=" * 60)

    for i, query in enumerate(test_queries, 1):
        print(f"\n📋 Test Query {i}: '{query}'")
        print("-" * 50)

        # Basic RAG
        print("🔹 Basic RAG:")
        if basic_engine:
            # Uncomment when testing:
            # basic_response = basic_engine.query(query)
```

```
            # print(f"   Response: {str(basic_response)[:200]}...")
            print("   (Standard vector search + simple response)")

        # Advanced RAG (if implemented)
        print("\n🔸 Advanced RAG:")
        if components_status["Advanced Pipeline"]:
            # Uncomment when testing:
            # advanced_response = advanced_pipeline.query(query)
            # print(f"   Response: {advanced_response}")
            print("   (Filtered + TreeSummarize + Structured output)")
        else:
            print("   Complete the advanced pipeline function to test")

# Final status
print("\n" + "=" * 60)
print("🎯 Assignment Status:")
completed_count = sum(components_status.values())
total_count = len(components_status)

print(f"   Completed: {completed_count}/{total_count} components")

if completed_count == total_count:
    print("\n🎉 Congratulations! You've mastered Advanced RAG Techniques!")
    print("   ✅ Node postprocessors for result filtering")
    print("   ✅ Response synthesizers for better answers")
    print("   ✅ Structured outputs for reliable data")
    print("   ✅ Advanced pipelines combining all techniques")
    print("\n🚀 You're ready for production RAG systems!")
else:
    missing = total_count - completed_count
    print(f"\n📝 Complete {missing} more components to finish the assignment:")
    for component, status in components_status.items():
        if not status:
            print(f"   - {component}")

print("\n💡 Key learnings:")
print("   - Postprocessors improve result relevance and precision")
print("   - Different synthesizers work better for different query types")
print("   - Structured outputs enable reliable system integration")
print("   - Advanced techniques can be combined for production systems")
```

```
🚀 Advanced RAG Techniques Assignment - Final Test
============================================================

📊 Component Status:
   ✅ Basic Index
   ✅ Similarity Filter
   ✅ TreeSummarize
   ✅ Structured Output
   ✅ Advanced Pipeline
```

```
🔍 Creating basic query engine for comparison...

============================================================
🆚 COMPARISON: Basic vs Advanced RAG
============================================================

📋 Test Query 1: 'What are the key capabilities of AI agents?'
--------------------------------------------------
🔹 Basic RAG:
   (Standard vector search + simple response)

🔸 Advanced RAG:
   (Filtered + TreeSummarize + Structured output)

📋 Test Query 2: 'How do you evaluate agent performance metrics?'
--------------------------------------------------
🔹 Basic RAG:
   (Standard vector search + simple response)

🔸 Advanced RAG:
   (Filtered + TreeSummarize + Structured output)

📋 Test Query 3: 'Explain the benefits and challenges of multimodal AI systems'
--------------------------------------------------
🔹 Basic RAG:
   (Standard vector search + simple response)

🔸 Advanced RAG:
   (Filtered + TreeSummarize + Structured output)

============================================================
🎯 Assignment Status:
   Completed: 5/5 components

🎉 Congratulations! You've mastered Advanced RAG Techniques!
   ✅ Node postprocessors for result filtering
   ✅ Response synthesizers for better answers
   ✅ Structured outputs for reliable data
   ✅ Advanced pipelines combining all techniques

🚀 You're ready for production RAG systems!

💡 Key learnings:
   - Postprocessors improve result relevance and precision
   - Different synthesizers work better for different query types
   - Structured outputs enable reliable system integration
```