Day 4: RAG & Evaluation

Agentic AI Intensive Training

5-Day Program

Today's Agenda

Morning: Retrieval-Augmented Generation

- RAG fundamentals
- Vector databases
- Embeddings

Afternoon: Evaluation Frameworks

- Testing strategies
- Metrics & benchmarks
- Quality assurance

Evening: Hands-On

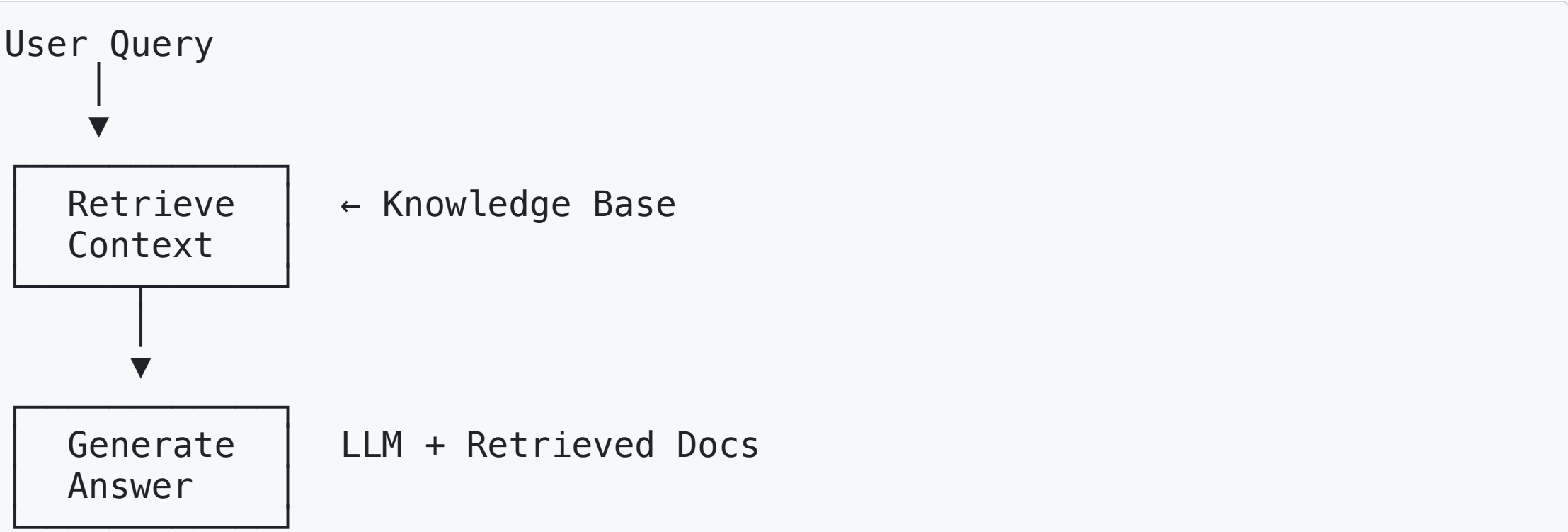
- Build RAG system

What is RAG?

Retrieval-Augmented Generation

Problem: LLMs have limited/outdated knowledge

Solution: Retrieve relevant info, then generate



Why RAG?

Benefits

- Access to up-to-date info
- Private/proprietary data
- Factual grounding
- Attribution/citations
- Cost-effective vs fine-tuning

Use Cases

- Documentation Q&A
- Customer support
- Research assistance
- Legal/medical queries

RAG vs Fine-Tuning

Aspect	RAG	Fine-Tuning
Data updates	Easy	Requires retraining
Setup cost	Low	High
Accuracy	Good with context	Better patterns
Citations	Easy	Hard

Best: Combine both approaches

RAG Pipeline Overview

```
1. Indexing (One-time)
   Documents → Chunks → Embeddings → Vector DB

2. Retrieval (Per Query)
   Query → Embedding → Search → Top K docs

3. Generation
   Query + Docs → LLM → Answer
```

Document Chunking

Why Chunk?

- Token limits
- Focused context
- Better retrieval precision

Strategies

1. Fixed size (e.g., 500 tokens)
2. Sentence boundaries
3. Paragraph-based
4. Semantic (topic changes)

Python: Chunking Implementation

```python
def chunk_text(text, chunk_size=500, overlap=50):
    """Split text into overlapping chunks"""
    words = text.split()
    chunks = []

    for i in range(0, len(words), chunk_size - overlap):
        chunk = " ".join(words[i:i + chunk_size])
        chunks.append(chunk)

    return chunks

# Usage
doc = "Long document text..."
chunks = chunk_text(doc)
print(f"Created {len(chunks)} chunks")
```

TypeScript: Chunking

```typescript
function chunkText(
  text: string,
  chunkSize = 500,
  overlap = 50
): string[] {
  const words = text.split(/\s+/);
  const chunks: string[] = [];

  for (let i = 0; i < words.length; i += chunkSize - overlap) {
    const chunk = words.slice(i, i + chunkSize).join(' ');
    chunks.push(chunk);
  }

  return chunks;
}

// Usage
const doc = "Long document text...";
const chunks = chunkText(doc);
console.log(`Created ${chunks.length} chunks`);
```

Embeddings

What are Embeddings?

- Vector representation of text
- Semantic meaning encoded
- Similar text → similar vectors

Common Models

- OpenAI: text-embedding-3-small/large
- Open source: sentence-transformers
- Specialized: domain-specific models

Dimensions: 384 to 3072+

Python: Generate Embeddings

```python
from openai import OpenAI

client = OpenAI()

def get_embedding(text, model="text-embedding-3-small"):
    response = client.embeddings.create(
        input=text,
        model=model
    )
    return response.data[0].embedding

# Usage
text = "Machine learning is fascinating"
embedding = get_embedding(text)
print(f"Embedding size: {len(embedding)}")
# Output: 1536 dimensions
```

TypeScript: Generate Embeddings

```typescript
import OpenAI from "openai";

const openai = new OpenAI();

async function getEmbedding(
  text: string,
  model = "text-embedding-3-small"
): Promise<number[]> {
  const response = await openai.embeddings.create({
    input: text,
    model: model
  });
  return response.data[0].embedding;
}

// Usage
const text = "Machine learning is fascinating";
const embedding = await getEmbedding(text);
console.log(`Embedding size: ${embedding.length}`);
```

Vector Similarity

Cosine Similarity

```
import numpy as np

def cosine_similarity(vec1, vec2):
    dot_product = np.dot(vec1, vec2)
    norm1 = np.linalg.norm(vec1)
    norm2 = np.linalg.norm(vec2)
    return dot_product / (norm1 * norm2)

# Compare embeddings
sim = cosine_similarity(embedding1, embedding2)
print(f"Similarity: {sim:.3f}")  # 0.0 to 1.0
```

High similarity = semantically similar

Vector Databases

Purpose: Efficient similarity search

Popular Options

Database	Type	Best For
Pinecone	Cloud	Scale
Weaviate	Open/Cloud	Features
Chroma	Embedded	Dev/Test
FAISS	Library	Speed
Qdrant	Open/Cloud	Balance

Chroma DB Setup

```
import chromadb

# Create client
client = chromadb.Client()

# Create collection
collection = client.create_collection(
    name="my_docs",
    metadata={"description": "Document collection"}
)

# Add documents
collection.add(
    documents=["Text 1", "Text 2", "Text 3"],
    metadatas=[{"source": "doc1"}, ...],
    ids=["id1", "id2", "id3"]
)

# Query
results = collection.query(
    query_texts=["search query"],
    n_results=5
)
```

Pinecone Setup

```python
from pinecone import Pinecone

# Initialize
pc = Pinecone(api_key="...")
index = pc.Index("my-index")

# Upsert vectors
index.upsert(vectors=[
    ("id1", embedding1, {"text": "...", "source": "..."}),
    ("id2", embedding2, {"text": "...", "source": "..."})
])

# Query
results = index.query(
    vector=query_embedding,
    top_k=5,
    include_metadata=True
)

for match in results['matches']:
    print(f"Score: {match['score']}")
    print(f"Text: {match['metadata']['text']}")
```

Building RAG System

Step 1: Index Documents

```
def index_documents(docs, collection):
    for i, doc in enumerate(docs):
        # Chunk
        chunks = chunk_text(doc)

        # Embed
        embeddings = [get_embedding(c) for c in chunks]

        # Store
        collection.add(
            documents=chunks,
            embeddings=embeddings,
            ids=[f"doc{i}_chunk{j}" for j in range(len(chunks))]
        )
```

RAG: Retrieval Step

```
def retrieve_context(query, collection, k=3):
    # Get query embedding
    query_embedding = get_embedding(query)

    # Search
    results = collection.query(
        query_embeddings=[query_embedding],
        n_results=k
    )

    # Extract documents
    docs = results['documents'][0]
    return docs
```

RAG: Generation Step

```
def generate_answer(query, context_docs):
    # Build prompt with context
    context = "\n\n".join(context_docs)

    prompt = f"""
Answer based on the following context.

Context:
{context}

Question: {query}

Answer:"""

    # Generate
    response = client.chat.completions.create(
        model="gpt-4o",
        messages=[{"role": "user", "content": prompt}]
    )

    return response.choices[0].message.content
```

Complete RAG Pipeline

```python
def rag_pipeline(query, collection):
    # 1. Retrieve relevant docs
    context_docs = retrieve_context(query, collection, k=3)

    # 2. Generate answer
    answer = generate_answer(query, context_docs)

    return {
        "answer": answer,
        "sources": context_docs
    }

# Usage
result = rag_pipeline(
    "What is machine learning?",
    collection
)
print(result["answer"])
print("\nSources:", result["sources"])
```

Advanced RAG: Hybrid Search

Combine multiple retrieval methods

```
def hybrid_search(query, collection):
    # 1. Semantic search (embeddings)
    semantic_results = collection.query(
        query_embeddings=[get_embedding(query)],
        n_results=10
    )

    # 2. Keyword search (BM25)
    keyword_results = bm25_search(query, collection)

    # 3. Combine scores (weighted)
    combined = merge_results(
        semantic_results,
        keyword_results,
        weights=[0.7, 0.3]
    )
```

Advanced RAG: Re-ranking

Improve retrieval quality

```python
from sentence_transformers import CrossEncoder

reranker = CrossEncoder('cross-encoder/ms-marco-MiniLM-L-6-v2')

def rerank_results(query, docs):
    # Score each doc
    pairs = [[query, doc] for doc in docs]
    scores = reranker.predict(pairs)

    # Sort by score
    ranked = sorted(
        zip(docs, scores),
        key=lambda x: x[1],
        reverse=True
    )

    return [doc for doc, score in ranked]
```

Advanced RAG: Query Expansion

Generate alternative queries

```
def expand_query(query):
    prompt = f"""
Generate 3 alternative phrasings of this query:
"{query}"

Format: one per line
"""
    response = llm(prompt)
    expanded = response.split('\n')
    return [query] + expanded

# Retrieve with all variants
def multi_query_retrieve(query, collection):
    queries = expand_query(query)
    all_docs = []
    for q in queries:
        docs = retrieve_context(q, collection, k=2)
```

Advanced RAG: Parent-Child

Store chunks, retrieve parents

```
# Index with hierarchy
collection.add(
    documents=chunks,
    embeddings=chunk_embeddings,
    metadatas=[{
        "parent_id": parent_doc_id,
        "chunk_index": i
    } for i in range(len(chunks))]
)

# Retrieve chunks, return full docs
def parent_document_retrieve(query, collection):
    # Find matching chunks
    results = collection.query(query_embeddings=[...], n_results=5)

    # Get parent IDs
    parent_ids = {m['parent_id'] for m in results['metadatas'][0]}
```

LangChain RAG

```
from langchain.vectorstores import Chroma
from langchain.embeddings import OpenAIEmbeddings
from langchain.chains import RetrievalQA
from langchain_openai import ChatOpenAI

# Create vector store
embeddings = OpenAIEmbeddings()
vectorstore = Chroma.from_documents(
    documents=docs,
    embedding=embeddings
)

# Create retrieval chain
qa_chain = RetrievalQA.from_chain_type(
    llm=ChatOpenAI(model="gpt-4o"),
    chain_type="stuff",
    retriever=vectorstore.as_retriever(search_kwargs={"k": 3})
)

# Query
answer = qa_chain.run("What is machine learning?")
```

LlamaIndex RAG

```
from llama_index import VectorStoreIndex, SimpleDirectoryReader
from llama_index.llms import OpenAI

# Load documents
documents = SimpleDirectoryReader('docs/').load_data()

# Create index
index = VectorStoreIndex.from_documents(documents)

# Query
query_engine = index.as_query_engine(
    llm=OpenAI(model="gpt-4o"),
    similarity_top_k=3
)

response = query_engine.query("What is machine learning?")
print(response)
```

Evaluation: Why It Matters

Challenges

- Subjective quality
- Many "correct" answers
- Context-dependent
- Hard to automate

Need For

- Confidence in production
- Iteration guidance
- Regression detection
- Performance tracking

Evaluation Categories

1. Component-Level

- Retrieval quality
- Generation quality
- End-to-end

2. Metrics

- Accuracy
- Relevance
- Faithfulness
- Completeness

3. Methods

Retrieval Evaluation

Metrics

Metric	Definition	Good Value
Precision@K	Relevant in top K	>0.8
Recall@K	Found relevant	>0.7
MRR	Mean reciprocal rank	>0.8
NDCG	Ranking quality	>0.7

Python: Retrieval Metrics

```python
def precision_at_k(retrieved, relevant, k):
    """Precision in top K results"""
    top_k = retrieved[:k]
    relevant_in_k = len(set(top_k) & set(relevant))
    return relevant_in_k / k

def recall_at_k(retrieved, relevant, k):
    """Recall in top K results"""
    top_k = retrieved[:k]
    relevant_in_k = len(set(top_k) & set(relevant))
    return relevant_in_k / len(relevant)

# Example
retrieved = ["doc1", "doc5", "doc2", "doc8"]
relevant = ["doc1", "doc2", "doc3"]

print(f"P@3: {precision_at_k(retrieved, relevant, 3):.2f}")
print(f"R@3: {recall_at_k(retrieved, relevant, 3):.2f}")
```

Generation Evaluation

Metrics

- **Faithfulness**: Answer grounded in context?
- **Relevance**: Addresses the question?
- **Completeness**: Covers all aspects?
- **Conciseness**: No unnecessary info?

Methods

- Exact match (limited use)
- Semantic similarity
- LLM-as-judge
- Human annotation

LLM-as-Judge

Use LLM to evaluate outputs

```
def llm_judge(question, context, answer):
    prompt = f"""
Evaluate this answer on a scale of 1-5:

Question: {question}
Context: {context}
Answer: {answer}

Criteria:
1. Faithfulness (grounded in context)
2. Relevance (addresses question)
3. Completeness

Provide scores as JSON: {{"faithful": X, "relevant": X, ...}}
"""

    response = llm(prompt)
```

Python: Semantic Similarity

```python
from sklearn.metrics.pairwise import cosine_similarity

def semantic_similarity_score(predicted, reference):
    # Get embeddings
    pred_emb = get_embedding(predicted)
    ref_emb = get_embedding(reference)

    # Compute similarity
    similarity = cosine_similarity(
        [pred_emb],
        [ref_emb]
    )[0][0]

    return similarity

# Example
pred = "Paris is the capital of France"
ref = "The capital of France is Paris"
score = semantic_similarity_score(pred, ref)
print(f"Similarity: {score:.3f}")
```

RAGAS Framework

RAG Assessment Framework

```
from ragas import evaluate
from ragas.metrics import (
    faithfulness,
    answer_relevancy,
    context_precision,
    context_recall
)

# Prepare dataset
dataset = {
    "question": [...],
    "answer": [...],
    "contexts": [...],
    "ground_truth": [...]
}

# Evaluate
results = evaluate(
    dataset,
    metrics=[
        faithfulness,
        answer_relevancy,
        context_precision,
        context_recall
    ]
```

Test Dataset Creation

```
test_cases = [
    {
        "question": "What is machine learning?",
        "contexts": ["ML is a subset of AI...", ...],
        "ground_truth": "Machine learning is...",
        "metadata": {"difficulty": "easy", "topic": "basics"}
    },
    {
        "question": "Explain backpropagation",
        "contexts": ["Backprop is an algorithm...", ...],
        "ground_truth": "Backpropagation is...",
        "metadata": {"difficulty": "hard", "topic": "technical"}
    }
]

# Diverse coverage
# - Easy, medium, hard
# - Different topics
# - Edge cases
# - Common failures
```

Unit Testing RAG Components

```python
import pytest

def test_chunking():
    text = "A" * 1000
    chunks = chunk_text(text, chunk_size=100)
    assert len(chunks) > 0
    assert all(len(c) <= 100 for c in chunks)

def test_embedding_generation():
    text = "Test text"
    emb = get_embedding(text)
    assert len(emb) == 1536  # Expected dimension
    assert all(isinstance(x, float) for x in emb)

def test_retrieval():
    query = "machine learning"
    docs = retrieve_context(query, collection, k=3)
    assert len(docs) == 3
    assert all(isinstance(d, str) for d in docs)
```

Integration Testing

```
def test_rag_pipeline():
    # Setup
    docs = ["ML is...", "AI is...", "DL is..."]
    collection = setup_collection(docs)

    # Test query
    result = rag_pipeline("What is ML?", collection)

    # Assertions
    assert "answer" in result
    assert "sources" in result
    assert len(result["sources"]) > 0
    assert "machine learning" in result["answer"].lower()

def test_rag_with_no_results():
    result = rag_pipeline("asdfasdfasdf", collection)
    assert "cannot answer" in result["answer"].lower()
```

Performance Testing

```python
import time

def test_latency():
    queries = ["query1", "query2", "query3"]

    latencies = []
    for query in queries:
        start = time.time()
        rag_pipeline(query, collection)
        latency = time.time() - start
        latencies.append(latency)

    avg_latency = sum(latencies) / len(latencies)
    assert avg_latency < 2.0  # 2 second SLA

def test_token_usage():
    result = rag_pipeline_with_tracking("query", collection)
    assert result["tokens"] < 2000  # Budget limit
```

Regression Testing

```
# Store baseline results
baseline_results = {
    "query1": {"score": 0.85, "answer": "..."},
    "query2": {"score": 0.90, "answer": "..."}
}

def test_no_regression():
    for query, expected in baseline_results.items():
        result = evaluate_rag(query, collection)

        # Check score hasn't dropped
        assert result["score"] >= expected["score"] - 0.05

        # Check semantic similarity
        similarity = semantic_similarity_score(
            result["answer"],
            expected["answer"]
        )
        assert similarity > 0.85
```

A/B Testing Framework

```
class ABTest:
    def __init__(self, variant_a, variant_b):
        self.variant_a = variant_a
        self.variant_b = variant_b
        self.results_a = []
        self.results_b = []

    def run(self, query):
        # Random assignment
        if random.random() < 0.5:
            result = self.variant_a.process(query)
            self.results_a.append(result)
        else:
            result = self.variant_b.process(query)
            self.results_b.append(result)
        return result

    def analyze(self):
        avg_a = np.mean([r["score"] for r in self.results_a])
        avg_b = np.mean([r["score"] for r in self.results_b])
        return {"variant_a": avg_a, "variant_b": avg_b}
```

Human Evaluation

When to Use

- Final quality check
- Subjective aspects (tone, style)
- Edge cases
- Gold standard creation

Process

1. Sample outputs
2. Create rubric
3. Multiple annotators
4. Measure agreement
5. Aggregate scores

Annotation Interface Example

```
def annotation_interface():
    sample = get_random_sample()

    print(f"Question: {sample['question']}")
    print(f"Context: {sample['context']}")
    print(f"Answer: {sample['answer']}")
    print("\nRate 1-5:")

    faithful = int(input("Faithfulness: "))
    relevant = int(input("Relevance: "))
    complete = int(input("Completeness: "))

    return {
        "sample_id": sample["id"],
        "faithful": faithful,
        "relevant": relevant,
        "complete": complete,
        "annotator": "user123"
    }
```

Continuous Evaluation

```
class ContinuousEvaluator:
    def __init__(self, test_suite):
        self.test_suite = test_suite
        self.history = []

    def evaluate_version(self, version, system):
        results = []
        for test in self.test_suite:
            result = system.process(test["query"])
            score = evaluate(result, test["expected"])
            results.append(score)

        avg_score = np.mean(results)
        self.history.append({
            "version": version,
            "score": avg_score,
            "timestamp": time.time()
        })
        return avg_score
```

Monitoring in Production

Key Metrics

- Response latency
- Token usage
- Error rates
- User feedback
- Retrieval quality

```
from prometheus_client import Counter, Histogram

query_counter = Counter('rag_queries_total', 'Total queries')
latency_histogram = Histogram('rag_latency_seconds', 'Latency')

@latency_histogram.time()
def rag_query(query):
    query_counter.inc()
```

Error Analysis

```
def analyze_failures(results):
    failures = [r for r in results if r["score"] < 0.5]

    # Categorize
    categories = {
        "retrieval_failure": [],
        "generation_failure": [],
        "ambiguous_query": []
    }

    for failure in failures:
        category = classify_failure(failure)
        categories[category].append(failure)

    # Report
    for cat, cases in categories.items():
        print(f"{cat}: {len(cases)} cases")
        print(f"Example: {cases[0] if cases else 'None'}")
```

Optimization Strategies

Based on Evaluation Results

1. **Low Retrieval Recall**
    - Improve chunking strategy
    - Try hybrid search
    - Expand query
2. **Low Faithfulness**
    - Better prompting
    - Stronger grounding
    - Add citations
3. **High Latency**

Workshop: Build RAG System

Goal: Document Q&A system

Steps

1. Collect documents (markdown files)
2. Chunk and embed
3. Store in vector DB
4. Implement retrieval
5. Generate answers
6. Evaluate quality

Workshop: Evaluation Suite

Goal: Test RAG system quality

Tasks

1. Create test dataset (10+ questions)
2. Define gold standard answers
3. Implement metrics
4. Run evaluation
5. Analyze results
6. Iterate improvements

Exercise 1: Basic RAG

```
def exercise1():
    # Load documents
    docs = load_docs("./data/")

    # Create collection
    collection = create_collection(docs)

    # Test query
    result = rag_pipeline(
        "What are transformers?",
        collection
    )

    print(result["answer"])
    print("Sources:", len(result["sources"]))
```

Exercise 2: Evaluation

```
def exercise2():
    test_cases = [
        {"q": "What is AI?", "expected": "..."},
        {"q": "Explain neural networks", "expected": "..."},
        # Add more...
    ]

    scores = []
    for test in test_cases:
        result = rag_pipeline(test["q"], collection)
        score = evaluate_answer(
            result["answer"],
            test["expected"]
        )
        scores.append(score)

    print(f"Average score: {np.mean(scores):.2f}")
```

Exercise 3: Optimization

Compare retrieval strategies

```
def exercise3():
    query = "What are the applications of AI?"

    # Strategy 1: Simple semantic search
    result1 = rag_with_semantic_search(query)

    # Strategy 2: Hybrid search
    result2 = rag_with_hybrid_search(query)

    # Strategy 3: Query expansion
    result3 = rag_with_query_expansion(query)

    # Evaluate and compare
    scores = [evaluate(r) for r in [result1, result2, result3]]
    print("Best strategy:", ["semantic", "hybrid", "expansion"][np.argmax(scores)])
```

Best Practices Summary

RAG Design

- Appropriate chunk size (test different sizes)
- Overlap between chunks
- Good metadata tracking
- Efficient vector DB
- Monitor retrieval quality

Evaluation

- Diverse test cases
- Multiple metrics
- Automated + human eval
- Continuous monitoring

Common Pitfalls

RAG Issues

- Chunks too large/small
- Poor retrieval precision
- Context window overflow
- Outdated embeddings
- No source attribution

Evaluation Issues

- Insufficient test coverage
- Biased test data
- Ignoring edge cases
- No baseline comparison

Production Checklist

Before Launch

- [ ] Comprehensive test suite
- [ ] Retrieval evaluation >80%
- [ ] Latency under SLA
- [ ] Error handling
- [ ] Monitoring setup
- [ ] Fallback mechanisms
- [ ] Cost projections
- [ ] Security review

Advanced Topics

Beyond Basics

- Multi-modal RAG (images, tables)
- Agents with RAG
- Streaming RAG responses
- Federated search
- Privacy-preserving RAG
- Graph-based RAG

Day 5: Production deployment

Resources

Tools

- Vector DBs: Pinecone, Weaviate, Chroma
- Frameworks: LangChain, LlamaIndex
- Evaluation: RAGAS, DeepEval

Papers

- "Retrieval-Augmented Generation for NLP"
- "Lost in the Middle" (context ordering)
- "Precise Zero-Shot Dense Retrieval"

Q&A

Questions?

Tomorrow: Day 5 - Production Systems

- Deployment strategies
- Scaling considerations
- Security & compliance
- Monitoring & ops

Thank You

Almost there!

Homework

- Complete RAG system
- Run evaluation suite
- Analyze failure cases
- Prepare for production discussion

Final day tomorrow!