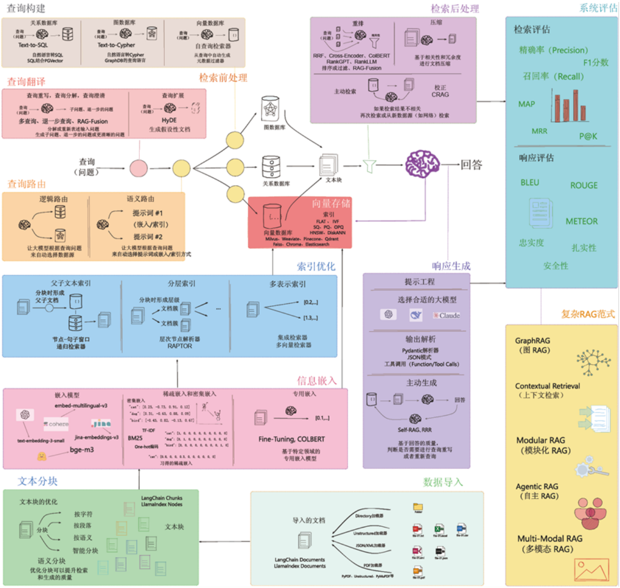
RAG is All You Need: A Comprehensive Guide to Retrieval-Augmented Generation

Zhanghao Chen

HKUST(GZ)
IPE Thrust

August 2025

Blueprint



查询构建
关系数据库
Text-to-SQL
图数据库
Text-to-Cypher
向量数据库
自查询检索器
查询翻译
检索前处理
查询重写、查询分解、查询澄清
多查询、退一步查询、RAG-Fusion
查询扩展
HyDE
生成假设性文档
查询
（问题）
查询路由
逻辑路由
语义路由
提示词 #1
（嵌入/索引）
提示词 #2
图数据库
关系数据库
文本块
回答
向量存储
索引
向量数据库
检索后处理
重排
压缩
RRF、Cross-Encoder、ColBERT
RankGPT、RankLLM
排序或过滤、RAG-Fusion
基于相关性和冗余度
进行文档压缩
主动检索
校正
CRAG
系统评估
检索评估
精确率（Precision）
F1分数
召回率（Recall）
MAP
MRR
P@K
响应评估
BLEU
ROUGE
METEOR
忠实度
扎实性
安全性
索引优化
父子文本索引
分层索引
多表示索引
RAPTOR
响应生成
提示工程
选择合适的大模型
Claude
输出解析
主动生成
Self-RAG, RRR
复杂RAG范式
GraphRAG
（图 RAG）
Contextual Retrieval
（上下文检索）
Modular RAG
（模块化 RAG）
Agentic RAG
（自主 RAG）
Multi-Modal RAG
（多模态 RAG）
信息嵌入
嵌入模型
embed-multilingual-v3
cohere
jina
text-embedding-3-small
jina-embeddings-v3
bge-m3
稀疏嵌入和密集嵌入
TF-IDF
BM25
专用嵌入
Fine-Tuning, COLBERT
文本分块
文本块的优化
LangChain Chunks
LlamaIndex Nodes
按字符
按段落
按语义
智能分块
文本块
分块
语义分块
数据导入
导入的文档
LangChain Documents
LlamaIndex Documents

Outline

Introduction to RAG Systems

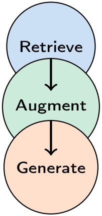
What is RAG?

Definition

Retrieval-Augmented Generation (RAG) combines information retrieval from external knowledge bases with text generation using large language models.

Three Core Steps:

1. **Retrieve:** Search relevant documents
2. **Augment:** Combine context with query
3. **Generate:** Produce grounded responses



Key Benefit: Enables dynamic, up-to-date, factually grounded responses

Why RAG Matters

Traditional LLM Challenges:

- Static training data
- Expensive retraining
- Hallucinations
- Limited domain knowledge

RAG Solutions:

- Dynamic knowledge access
- Cost-effective updates
- Factual grounding
- Domain specialization

Key Insight: RAG transforms LLMs from static knowledge repositories into dynamic information processing systems.

Comparison

Aspect	Pure LLM	Fine-tuning	RAG
Knowledge Updates	No	No	Yes
Cost	Low	High	Medium
Accuracy	Medium	High	High
Flexibility	Low	Medium	High
Implementation	Easy	Hard	Medium

- RAG offers the best balance of accuracy, flexibility, and cost
- Enables real-time knowledge integration without model retraining

Real-World Applications

Enterprise Use Cases:

- Customer support chatbots
- Internal knowledge management
- Legal document analysis
- Technical documentation

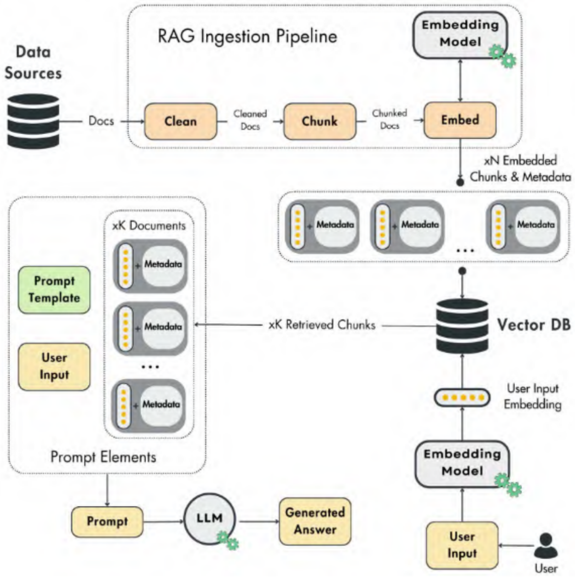
Research & Education:

- Scientific literature review
- Educational Q&A systems
- Research assistance
- Academic writing support

Real Case: Microsoft Copilot uses RAG to retrieve and summarize web content, providing users with accurate, up-to-date information from multiple sources in real-time.

Naive RAG

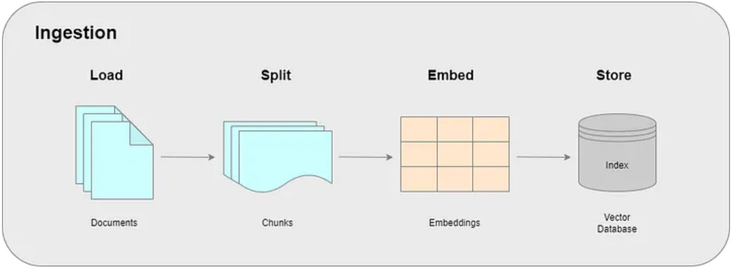
Naive RAG Architecture



Naive RAG: Three Main Components

1. **Document Ingestion Pipeline**
    - Text extraction and chunking
    - Embedding generation
    - Vector storage
2. **Retrieval Pipeline**
    - Query embedding
    - Similarity search
    - Top-k document selection
3. **Generation Pipeline**
    - Context integration
    - Prompt construction
    - Response generation

Ingestion



- **Text Extraction & Chunking:** Break documents into manageable pieces & remove all invalid characters
- **Embedding Generation:** Convert text chunks into vector representations
- **Vector Storage:** Store embeddings in searchable database

Text Extraction Strategies

Document Types:

- PDF documents
- Web pages (HTML)
- Office documents
- Plain text files
- Structured data (JSON, XML)

Extraction Tools:

- PyPDF2, pdfplumber
- BeautifulSoup, Scrapy
- python-docx, openpyxl
- Unstructured.io
- Custom parsers

Challenge: Preserving document structure and metadata during extraction is critical for context understanding.

Text Cleaning Example

```python
import re

def clean_text(text: str) -> str:
    # Remove extra whitespace
    text = re.sub(r'\s+', ' ', text)
    # Remove special characters but keep punctuation
    text = re.sub(r'[^\w\s\.\,\!\?\;\:]', '', text)
    # Remove very short lines (likely artifacts)
    lines = [line for line in text.split('\n')
             if len(line.strip()) > 10]
    return '\n'.join(lines).strip()
```

- Remove formatting artifacts and normalize whitespace
- Preserve semantic structure

Chunking Strategies

Strategy	Method
Fixed Size	Character/token count
Recursive Character	Multi-level separators
Document-Specific	Structure-aware splitting
Semantic	Embedding-based similarity
Agentic	LLM-driven decisions

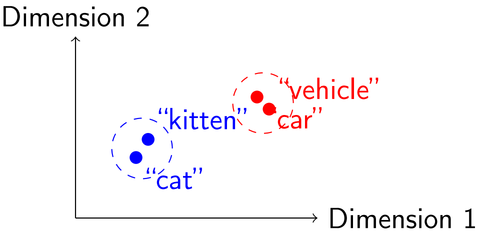
- **Chunk size:** Balance between context and specificity
- **Overlap:** Prevent information loss at boundaries
- **Metadata:** Preserve source information and structure

LangChain Implementation: Use `CharacterTextSplitter` for fixed size, `RecursiveCharacterTextSplitter` for recursive, and `SemanticChunker` for semantic chunking.

https://medium.com/@aminajavaid30/building-a-rag-system-the-data-ingestion-pipeline-d04235fd17ea

Embeddings

- **Definition:** Dense vector representations of text that capture semantic meaning
- **Purpose:** Enable mathematical operations on text for similarity search
- **Properties:**
  - Similar texts have similar vectors
  - Vectors can be compared using distance metrics



Popular Embedding Models (2025.08)

Model	Dims	Tokens	Best For
OpenAI text-embedding-3-large	3072	8191	General purpose
Google Gemini-embedding-001	3072	2048	Multilingual, flexible
Qwen3-Embedding-8B	4096	32768	Long context, multilingual
BGE-M3	1024	8192	Multilingual, hybrid
E5-Mistral-7B-instruct	4096	32768	High performance

- **Trade-offs:** Accuracy vs. Speed vs. Cost vs. Deployment
- **Customization:** Fine-tuned models for specialized domains
- **Consistency:** Same model for indexing and querying

Embedding Generation Example

```python
from sentence_transformers import SentenceTransformer

# Load model and generate embeddings
model = SentenceTransformer('all-MiniLM-L6-v2')
texts = ["The cat sat on the mat",
         "A feline rested on the rug"]
embeddings = model.encode(texts)

# Calculate similarity
from sklearn.metrics.pairwise import cosine_similarity
similarity = cosine_similarity(embeddings)[0][1]
print(f"Similarity: {similarity:.3f}")
```

Output:

```
Embedding shape: (2, 384)
Similarity: 0.564
```

- Each text becomes a 384-dimensional vector
- Cosine similarity measures semantic closeness

MTEB Embedding Evaluation Metrics

Task Category	Description	Example Datasets	Metric
Classification	Text classification tasks	Amazon, Banking77	Accuracy
Clustering	Document clustering	ArxivClustering, Reddit	V-measure
Pair Classification	Binary text pair classification	Sprint, TwitterSemEval	AP
Reranking	IR reranking	AskUbuntu, MindSmall	MAP
Retrieval	Document retrieval	MSMARCO, NQ	nDCG@10
STS	Semantic similarity	STS-B, SICK-R	Spearman
Summarization	Document summarization	SummEval	Spearman
BitextMining	Parallel sentence mining	BUCC, Tatoeba	F1 Score

- **MTEB:** Massive Text Embedding Benchmark with 8 task categories

`https://huggingface.co/spaces/mteb/leaderboard`

Sparse vs Dense Embeddings & BM25

Aspect	**Sparse Embeddings**	**Dense Embeddings**	**BM25**
Representation	High-dimensional sparse vectors	Low-dimensional dense vectors	Term frequency based
Interpretability	High (explicit terms)	Low (latent features)	High (term weights)
Semantic Understanding	Limited	Strong	None
Exact Match	Excellent	Poor	Excellent
Conceptual Match	Poor	Excellent	Poor
Storage	Efficient (sparse)	More storage needed	Minimal
Computation	Fast	Moderate	Very fast
Training Required	No	Yes	No

- **Sparse:** TF-IDF, SPLADE - good for keyword matching
- **Dense:** BERT, Sentence-BERT - captures semantic meaning
- **BM25:** Classic probabilistic ranking function
- **Hybrid:** Combine all three for optimal performance

Vector Databases

Traditional Databases:

- Exact match queries
- SQL-based operations
- Structured data focus
- Limited similarity search

Vector Databases:

- Similarity-based search
- High-dimensional vectors
- Semantic understanding
- Optimized for ML workloads

Key Capability: Vector databases enable fast approximate nearest neighbor (ANN) search across millions of high-dimensional vectors.

Vector Database Comparison

Database	Type	Scalability	Ease of Use	Cost
Pinecone	Cloud	High	High	Medium
Weaviate	Self-hosted/Cloud	High	Medium	Low-Med
Chroma	Self-hosted	Medium	High	Low
Qdrant	Self-hosted/Cloud	High	Medium	Low-Med
Milvus (Preferred)	Self-hosted/Cloud	Very High	Low	Low
FAISS	Library	Medium	Low	Free

- **Cloud vs. Self-hosted:** Trade-off between convenience and control
- **Scalability:** Consider your expected data volume and query load
- **Features:** Metadata filtering, hybrid search, multi-tenancy

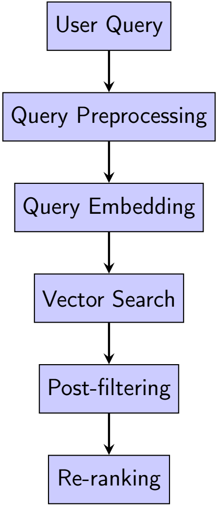
Vector Search Algorithms

- **Exact Search (Brute Force):**
  - Compare query vector with all stored vectors
  - Guaranteed accuracy but slow for large datasets
  - O(n) complexity
- **Approximate Nearest Neighbor (ANN):**
  - Trade accuracy for speed
  - Various algorithms: HNSW, IVF, LSH
  - Sub-linear complexity
- **Hierarchical Navigable Small World (HNSW):**
  - Most popular ANN algorithm
  - Graph-based approach
  - Excellent recall-speed trade-off

Types of Retrieval

1. **Dense Retrieval (Semantic Search):**
   - Uses vector embeddings
   - Captures semantic similarity
   - Good for conceptual matches
2. **Sparse Retrieval (Keyword Search):**
   - Traditional TF-IDF, BM25
   - Exact term matching
   - Good for specific terms/names
3. **Hybrid Retrieval:**
   - Combines dense and sparse methods
   - Best of both worlds
   - More robust results

Retrieval Pipeline



Query Enhancement Techniques

- **Query Expansion:**
  - Add synonyms and related terms
  - Use LLM to generate alternative phrasings
  - Domain-specific term expansion
- **Query Decomposition:**
  - Break complex queries into sub-questions
  - Retrieve for each sub-question separately
  - Combine results intelligently
- **Hypothetical Document Embeddings (HyDE):**
  - Generate hypothetical answer to query
  - Use answer embedding for retrieval
  - Often more effective than query embedding

Retrieval Implementation Example

```python
import chromadb
from sentence_transformers import SentenceTransformer

# Initialize components
client = chromadb.Client()
collection = client.create_collection("documents")
model = SentenceTransformer('all-MiniLM-L6-v2')

# Add documents and query
documents = ["Document 1 text...", "Document 2 text..."]
embeddings = model.encode(documents)
collection.add(embeddings=embeddings.tolist(),
               documents=documents,
               ids=["doc_1", "doc_2"])

query_embedding = model.encode(["What is ML?"])
results = collection.query(
    query_embeddings=query_embedding.tolist(),
    n_results=5)
```

The Generation Component

- **Role:** Synthesize retrieved information into coherent responses
- **Input:** User query + retrieved context documents
- **Output:** Natural language response grounded in retrieved content
- **Challenges:**
    - Context length limitations
    - Information synthesis
    - Maintaining factual accuracy
    - Handling conflicting information

Key Principle: The LLM should act as an intelligent synthesizer, not a creative writer, when using RAG.

Prompt Engineering for RAG

- **System Prompt:** Define the AI's role and behavior
- **Context Integration:** How to present retrieved documents
- **Instructions:** Specific guidelines for response generation
- **Output Format:** Structure the desired response format

Essential Elements:

1. Clear role definition
2. Context usage instructions
3. Accuracy requirements
4. Citation guidelines
5. Fallback behavior for insufficient context

RAG Prompt Template Example

```
You are a helpful AI assistant. Use the provided context
to answer the user's question accurately.

CONTEXT:
{retrieved_documents}

INSTRUCTIONS:
1. Base your answer primarily on the provided context
2. If context doesn't contain enough information, say so
3. Include relevant quotes when appropriate
4. Cite sources using [Source: document_name]
5. Be concise but thorough

USER QUESTION: {user_query}

ANSWER:
```

- Clear instructions prevent hallucination
- Citation requirements improve transparency
- Fallback behavior handles edge cases

Advanced Prompting Techniques

Chain-of-Thought:

- Step-by-step reasoning
- Improves complex queries
- Shows reasoning process

Few-Shot Examples:

- Provide example Q&A pairs
- Demonstrates desired format
- Improves consistency

Role-Based Prompting:

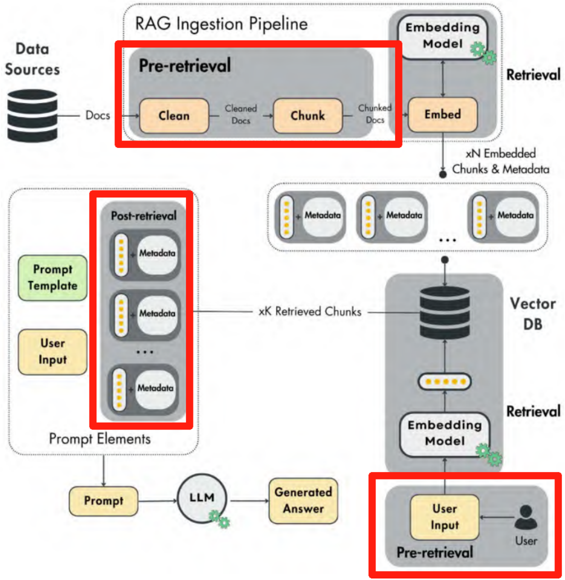
- Specific expert personas
- Domain-appropriate language
- Targeted expertise

Multi-Step Reasoning:

- Break down complex tasks
- Validate intermediate steps
- Reduce error propagation

Advanced RAG

Advanced RAG Architecture



Pre-Retrieval Optimization

Query Enhancement:

- Query expansion with synonyms
- Query rewriting for better matching
- Multi-query generation

Query Routing:

- Route to appropriate knowledge bases
- Domain-specific strategies

Query Classification:

- Intent detection
- Complexity assessment
- Response type prediction

Goal: Optimize queries before retrieval for better results.

Pre-Retrieval Techniques

HyDE (Hypothetical Document Embeddings):

- Generate hypothetical answer
- Use answer embedding for retrieval
- Often more effective than query embedding

Step-Back Prompting:

- Generate broader, conceptual questions
- Retrieve high-level context first
- Then focus on specific details

Query Decomposition:

- Break complex queries into sub-questions
- Retrieve for each sub-question
- Combine results intelligently

Multi-Vector Retrieval:

- Generate multiple query representations
- Retrieve using different embeddings
- Merge and rank results

Post-Retrieval Processing

- **Re-ranking:**
  - Cross-encoder models for better relevance scoring
  - LLM-based relevance assessment
  - Diversity-aware ranking
- **Context Compression:**
  - Remove irrelevant information
  - Summarize long documents
  - Extract key sentences and facts
- **Context Fusion:**
  - Merge information from multiple sources
  - Resolve contradictions
  - Create coherent context window

Goal: Refine and optimize retrieved content before feeding it to the generation model.

Reranking Techniques Overview

Reranking: Post-retrieval technique to improve document relevance ordering

Reciprocal Rank Fusion (RRF):

- Formula: $\text{RRF}(d) = \sum_{r \in R} \frac{1}{k + r(d)}$
- No training required
- Robust aggregation method

Cross-Encoder Models:

- Joint query-document encoding
- Higher accuracy, slower inference
- Examples: BERT, RoBERTa

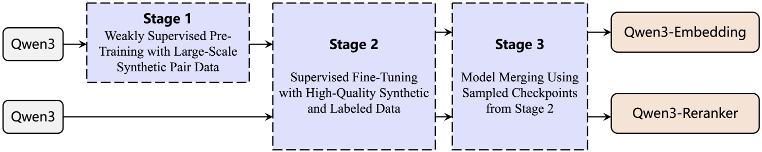
Cohere Rerank:

- Commercial API solution
- Multilingual support
- Domain-optimized

Benefits:

- Improved relevance ordering
- Better generation context
- Reduced noise in results

Qwen3: Unified Embedding & Reranking



Qwen3 Dual-Mode Architecture:

- **Embedding Mode:** Fast dense retrieval with shared parameters
- **Reranking Mode:** Cross-attention scoring with LM head
- **Unified Training:** Single model for both tasks
- **Performance:** SOTA on MTEB retrieval benchmark

Innovation: First model to unify embedding and reranking in one architecture, achieving both efficiency and accuracy.

Post-Retrieval Techniques

Lost-in-the-Middle Problem:

- LLMs perform worse on middle content
- Solution: Reorder by relevance
- Place most relevant at beginning/end

Context Window Management:

- Token budget allocation
- Intelligent truncation
- Sliding window approaches

Self-RAG:

- LLM evaluates retrieval necessity
- Adaptive retrieval based on confidence
- Self-correction mechanisms

CRAG (Corrective RAG):

- Evaluate retrieval quality
- Trigger web search if needed
- Correct and refine retrieved content

Graph RAG & Modular RAG

Graph RAG:

- Build knowledge graphs from documents
- Entity and relationship extraction
- Graph-based retrieval and reasoning

Benefits:

- Captures entity relationships
- Enables multi-hop reasoning
- Structured knowledge representation

Modular RAG:

- Flexible, composable components
- Mix and match different modules
- Adaptive pipeline design

Key Modules:

- Query processors
- Specialized retrievers
- Context processors
- Response generators

Future Direction: Evolution towards more sophisticated multimodal RAG systems.

Evaluation and Metrics

Why Evaluate RAG Systems?

- **Performance Optimization:** Identify bottlenecks and improvement areas
- **Component Analysis:** Understand which parts work well/poorly
- **Comparison:** Compare different approaches and configurations
- **Quality Assurance:** Ensure system meets requirements
- **Continuous Improvement:** Monitor performance over time

Challenge: RAG evaluation is complex because it involves both retrieval quality and generation quality.

Retrieval Evaluation Metrics

Traditional IR Metrics:

- **Precision@K:** Relevant docs in top K
- **Recall@K:** Coverage of relevant docs
- **Mean Reciprocal Rank (MRR):** Position of first relevant doc
- **NDCG:** Normalized discounted cumulative gain

RAG-Specific Metrics:

- **Context Relevance:** How relevant is retrieved context?
- **Context Recall:** Does context contain answer?
- **Context Precision:** How much context is relevant?
- **Answer Relevance:** Does answer address query?

- Requires ground truth relevance judgments
- Can be expensive to create comprehensive test sets

Generation Evaluation Metrics

Metric Type	Examples	Measures
Factual Accuracy	FactScore, FActCC	Correctness vs. source
Faithfulness	RAGAS Faithfulness	Grounding in context
Relevance	RAGAS Answer Relevance	Query-answer alignment
Completeness	Coverage metrics	Information completeness
Coherence	Human evaluation	Response quality

- **Automated metrics:** Scalable but may miss nuances
- **Human evaluation:** More accurate but expensive
- **LLM-as-judge:** Emerging approach using LLMs for evaluation

RAGAS Framework

- **Retrieval-Augmented Generation Assessment**
- Comprehensive evaluation framework for RAG systems
- Key metrics:
  - **Context Precision:** Relevant items in retrieved context
  - **Context Recall:** Ground truth in retrieved context
  - **Faithfulness:** Claims in answer supported by context
  - **Answer Relevance:** Answer addresses the question

$$\textbf{RAGAS Score} = \sqrt[4]{\text{Precision} \times \text{Recall} \times \text{Faithfulness} \times \text{Relevance}}$$

Evaluation Implementation

```python
from ragas import evaluate
from ragas.metrics import answer_relevancy, faithfulness

# Prepare evaluation dataset
eval_dataset = {
    'question': ['What is machine learning?'],
    'answer': ['ML is a subset of AI that enables
               systems to learn...'],
    'contexts': [['ML is a method of data analysis
                  that automates...']],
    'ground_truths': [['Machine learning is AI subset...']]
}

# Run evaluation
result = evaluate(eval_dataset,
                  metrics=[answer_relevancy, faithfulness])
print(f"Answer Relevancy: {result['answer_relevancy']}")
print(f"Faithfulness: {result['faithfulness']}")
```

RAG Tools & Frameworks

Popular RAG Frameworks

Framework	Type	Strengths	Best For
LangChain	Full-stack	Comprehensive, popular	Rapid prototyping
LlamaIndex	Data-focused	Document handling	Data-heavy applications
Haystack	Production-ready	Scalable, modular	Enterprise deployment
Chroma	Vector DB + RAG	Simple, integrated	Small to medium projects
Weaviate	Vector DB + RAG	GraphQL, hybrid search	Complex queries
Pinecone	Vector DB service	Managed, scalable	Production systems

- **Full-stack frameworks:** End-to-end RAG solutions
- **Specialized tools:** Focus on specific components
- **Cloud services:** Managed solutions with less setup

LangChain Framework Deep Dive

Key Components:

- Document loaders
- Text splitters
- Vector stores
- Retrievers
- Chains and agents

Advantages:

- Large ecosystem
- Many integrations
- Active community
- Extensive documentation

Use Case: Ideal for rapid prototyping and experimentation with different RAG configurations.

LangChain RAG Implementation

```
from langchain.document_loaders import TextLoader
from langchain.text_splitter import CharacterTextSplitter
from langchain.embeddings import OpenAIEmbeddings
from langchain.vectorstores import Chroma
from langchain.chains import RetrievalQA
from langchain.llms import OpenAI

# Load and process documents
loader = TextLoader('documents.txt')
documents = loader.load()
text_splitter = CharacterTextSplitter(
    chunk_size=1000, chunk_overlap=0)
texts = text_splitter.split_documents(documents)
```

LangChain RAG Implementation (continued)

```python
# Create vector store
embeddings = OpenAIEmbeddings()
vectorstore = Chroma.from_documents(texts, embeddings)

# Create RAG chain
qa_chain = RetrievalQA.from_chain_type(
    llm=OpenAI(), chain_type="stuff",
    retriever=vectorstore.as_retriever())

# Query
response = qa_chain.run("What is the main topic?")
print(response)
```

- Complete RAG pipeline in just a few lines
- Automatic document processing and indexing
- Built-in retrieval and generation

LlamaIndex Framework Deep Dive

- **Focus:** Data ingestion and indexing for LLM applications
- **Strengths:**
  - Advanced document parsing
  - Multiple index types
  - Query engines
  - Data connectors
- **Index Types:**
  - Vector Store Index
  - Tree Index
  - Keyword Table Index
  - Knowledge Graph Index

Use Case: Best for applications requiring sophisticated document understanding and multiple data sources.

Conclusion

- **RAG is transformative:** Bridges the gap between static LLMs and dynamic knowledge
- **Components matter:** Each part of the pipeline affects overall quality
- **Evaluation is key:** Measure what matters for your use case
- **Tools are maturing:** Rich ecosystem of frameworks and services
- **Future is bright:** Continuous improvements in all areas

RAG is indeed all you need for building intelligent, grounded AI systems!

References

1. Lewis, P., Perez, E., Piktus, A., et al. (2020). **Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks**. *Advances in Neural Information Processing Systems*, 33, 9459-9474.
2. Gao, Y., Xiong, Y., Gao, X., et al. (2023). **Retrieval-Augmented Generation for Large Language Models: A Survey**. *arXiv preprint arXiv:2312.10997*.
3. Huang, J. **RAG Practical Course**. GeekTime.
4. Iusztin, P., & Labonne, M. (2024). **LLM Engineer's Handbook: Master the Art of Engineering Large Language Models from Concept to Product**. Packt Publishing.
5. Mao, Y. **Large Language Model Fundamentals**.

Questions & Discussion

Thank you!

Questions & Discussion

Contact: zchen971@connect.hkust-gz.edu.cn

Slides available at: github.com/Zhanghao25/RAG