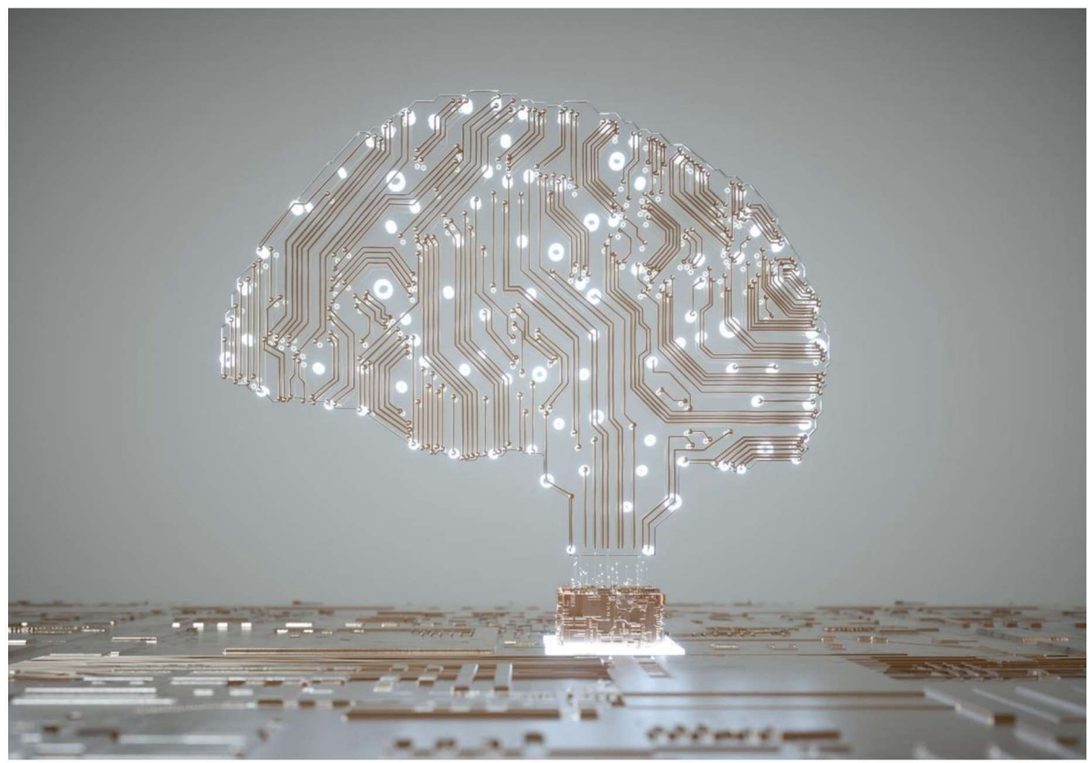
Retrieval Augmented Generation (RAG)

Deepan Sarkar



Why is RAG needed?

Drawbacks of LLM

Hallucination	• Outputs may include entirely **fictional** citations, people, or events that **appear convincingly real** • The model often **states incorrect information with high confidence**, making mistakes harder to detect • Hallucinations are **dangerous** in domains like healthcare, law, and education **where factual accuracy is critical**
Outdated information	• LLMs **cannot naturally incorporate breaking news, real-time changes, or live data** without augmentation • Users asking about recent events post-training cutoff will **receive outdated or speculative responses** • Model **knowledge becomes stale quickly in fast-moving fields** like tech, medicine, or geopolitics
Low efficiency in parameterizing knowledge	• **Updating** even a single **fact** (e.g., CEO name or regulation change) **requires costly fine-tuning or retraining** • Embedding knowledge directly into weights limits scalability for dynamically changing data • **Difficult to maintain consistency or remove outdated information** from the model's memory
Lack of in-depth knowledge in specialized domains	• The model **tends to generalize** or **oversimplify complex ideas** outside its training exposure • It **struggles with interpreting domain-specific** jargon, symbols, or reasoning frameworks • Errors in these areas are often subtle and misleading rather than obvious
Weak inferential capabilities	• LLMs may **fail to maintain logical consistency across multiple reasoning steps** • They are **poor at solving problems that require structured thinking**, such as deductive proofs or strategy planning • Inference tasks often require chaining multiple facts, which LLMs do unreliably
Lack of Explainability	• LLMs **do not cite sources** unless explicitly designed to do so via techniques like RAG • **Responses** are generated from internal embeddings, which **lack transparency or traceability** • **Hard to verify or audit outputs** in sensitive domains like finance, healthcare, or policy
Sensitivity to Prompting	• The **same question worded differently can yield vastly different answers** • **Prompt engineering** often becomes a **trial-and-error process** for users to get reliable results • **Malicious prompts** or **ambiguous phrasing** can trick the model into **undesired behavior**
Bias & Ethical Concerns	• Models trained on internet-scale data often **reflect stereotypes**, **offensive language**, or **historical injustices** • Responses may **inadvertently marginalize certain groups** or **reinforce biased viewpoints** • Managing and mitigating bias remains an open research and governance challenge
Context Length Limitations	• **Token limits** reduce how much context the model can retain, leading to **information loss in longer interactions** • **Long documents** are often **incompletely processed** or **incorrectly summarized** • **Threads across multiple conversations are easily forgotten** unless external memory is integrated

Practical Requirement of RAG

Domain-specific accurate answering

- External knowledge bases or document stores allow **better coverage of niche, technical, or regulated fields**
- Reduces the risk of hallucination by **grounding answers in verified domain content**
- Enables **fine-grained control** over terminology, tone, and compliance standards specific to each use case

Frequent updates of data

- **New content** (e.g., news, prices, regulations) can be **ingested instantly** into the retrieval layer
- **Eliminates the need for costly and time-consuming model fine-tuning**
- Ensures **time-sensitive answers** remain accurate and aligned with real-world changes

Traceability and explainability of generated content

- Users should be able to **view which documents or passages were retrieved and used**
- **Improves trust and reliability**, especially in high-stakes domains like healthcare, finance, or law
- Facilitates auditing, debugging, and manual verification of outputs

Controllable Cost

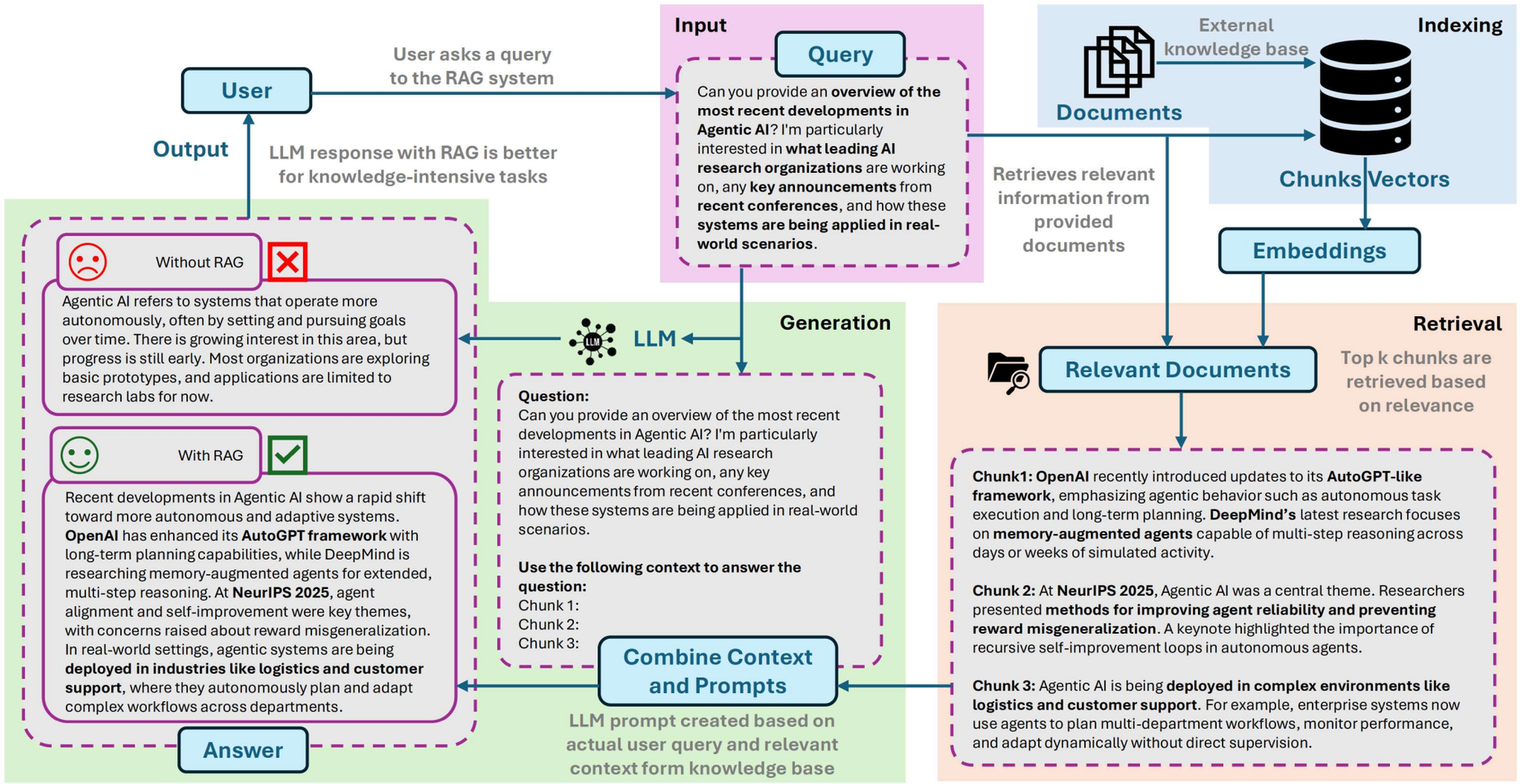
- **Avoids expensive retraining** of large language models for each new knowledge update
- Retrieval layer allows **lightweight inference with smaller or cheaper LLM backends**
- Enables **selective retrieval and compression** to minimize latency and API costs

Privacy protection of data

- Retrieved content can come from **private, access-controlled sources** rather than public web data
- Allows **compliance with data governance and regulatory frameworks** (e.g., GDPR, HIPAA)
- Keeps customer or business-specific **data siloed from model training and inference history**

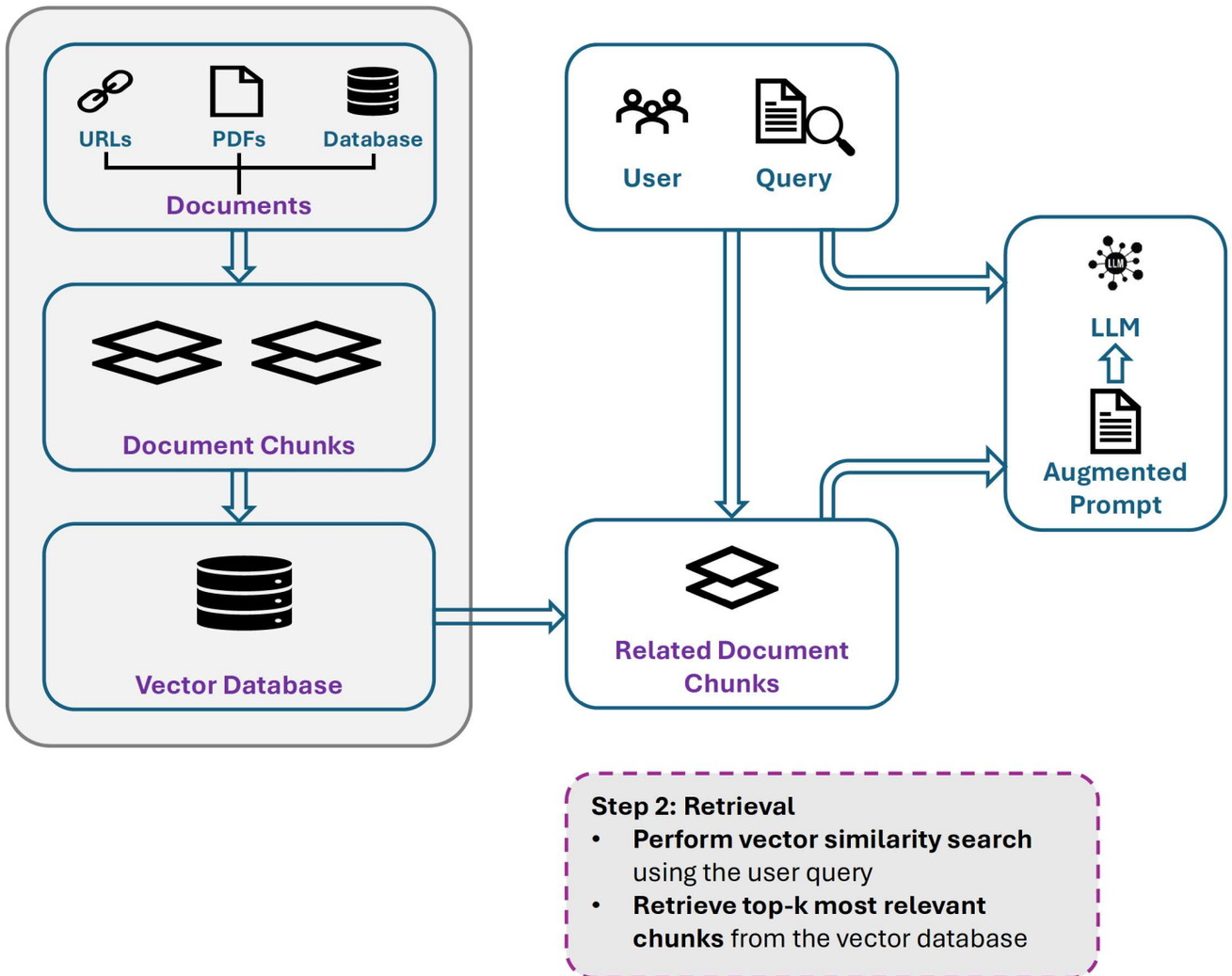
RAG Architecture

Basic RAG Architecture



Naive RAG

Step 1: Indexing
- **Split documents** into **smaller, manageable chunks**
- **Generate vector embeddings** for each chunk using an **encoding model**
- Store embeddings in a **vector database for efficient retrieval**



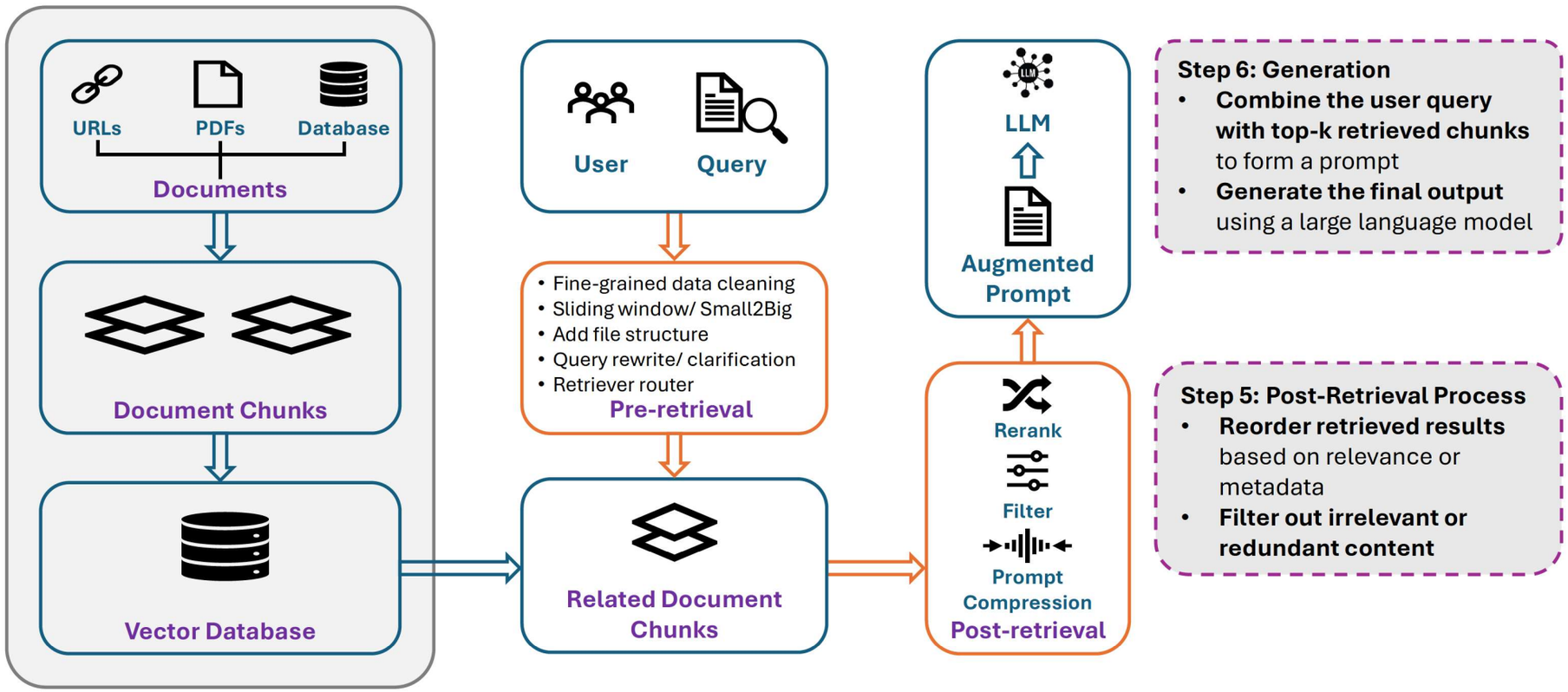
Step 3: Generation
- **Combine the user query with top-k retrieved chunks** to form a prompt
- **Generate the final output** using a large language model

Step 2: Retrieval
- **Perform vector similarity search** using the user query
- **Retrieve top-k most relevant chunks** from the vector database

Advanced RAG

Step 1: Indexing
- **Split documents** into **smaller, manageable chunks**
- **Generate vector embeddings** for each chunk using an **encoding model**
- Store embeddings in a **vector database for efficient retrieval**

Step 2: Optimizing Data Indexing
- Apply **sliding window** technique to preserve context
- Use **fine-grained segmentation** for better semantic coverage
- **Attach metadata** such as source, section title, or timestamp



Step 3: Pre-Retrieval Process
- **Define retrieval routes** based on user intent or context
- **Generate summaries** to support high-level filtering
- **Rewrite queries** for improved clarity and precision
- **Evaluate confidence scores** to guide relevance thresholds

Step 4: Retrieval
- **Perform vector similarity search** using the user query
- **Retrieve top-k most relevant chunks** from the vector database

Step 5: Post-Retrieval Process
- **Reorder retrieved results** based on relevance or metadata
- **Filter out irrelevant or redundant content**

Step 6: Generation
- **Combine the user query with top-k retrieved chunks** to form a prompt
- **Generate the final output** using a large language model

RAG Understanding

Key Questions of RAG

What to retrieve?

- Token
- Phrase
- Chunk
- Paragraph
- Entity
- Knowledge graph

When to retrieve?

- Single search
- Each token
- Every N tokens
- Adaptive search

How to use the retrieved information?

- Input/Data Layer
- Model/Intermediate Layer
- Output/Prediction Layer

What to retrieve?

Token: Retrieves at the level of individual words or subword units (e.g., BPE tokens).
Advantages:
- **Fine-grained matching** enables high flexibility across domains.
- Especially useful in **handling rare words** or **long-tail queries**.
- **Efficient computation** due to smaller unit size.

Disadvantages:
- Can be **semantically ambiguous**—tokens lack context.
- Requires **significant storage** and indexing complexity.
- May result in **fragmented and noisy results**, requiring extensive post-processing or aggregation.

Retrieval Unit	Precision	Recall	Storage	Speed	Best Use Cases
Token	Low	High	High	Fast	Cross-domain, long-tail
Phrase	Medium	High	Medium	Fast	Expression search, idioms
Chunk	Medium	High	Medium	Medium	Open-domain QA, summarization
Paragraph	High	Medium	Low	Slow	Legal, narrative, policy
Entity	High	Low	Low	Fast	Fact-based QA, KB lookup
KG	Very High	Low	Variable	Slow	Logical reasoning, structured tasks

Entity: Retrieves named or conceptual entities, potentially with metadata or descriptions.
Advantages:
- Supports **factual and structured queries** (e.g., "What is the capital of France?").
- Well-suited for **linking with structured data** or KBs.

Disadvantages:
- Requires **robust entity recognition and linking**.
- May struggle with **contextual disambiguation** ("Apple" the company vs. the fruit).
- Not ideal for open-ended or nuanced questions.

Phrase: Retrieves short, contiguous word sequences (e.g., n-grams).
Advantages:
- Captures **local semantic meaning** better than tokens.
- Can recall **common expressions or idioms** effectively.

Disadvantages:
- Still lacks broader context—easily retrieves **redundant or irrelevant** phrases.
- Suffers in **precision** for ambiguous queries.
- Not ideal for complex reasoning or synthesis tasks.

Chunk: Retrieves medium-sized blocks of text (e.g., 100–300 words), often pre-defined during document preprocessing.
Advantages:
- **High coverage**: useful in open-domain Q&A and summarization.
- Balances recall and computational cost well.
- Easy to implement and index.

Disadvantages:
- Often **overly broad**, pulling in **irrelevant or redundant** content.
- May not align cleanly with **semantic boundaries**, reducing relevance.
- Needs **strong reranking or filtering** downstream.

Paragraph: Retrieves complete paragraphs as atomic units.
Advantages:
- Maintains **semantic integrity**—better for tasks needing coherent context.
- Suitable for **narrative, legal, or policy** documents where ideas span multiple sentences.

Disadvantages:
- **Larger payloads** per retrieval—may exceed token limits.
- Precision can suffer if paragraphs are too verbose or contain multiple topics.
- Computational cost increases due to size.

Knowledge Graph (KG): Retrieves structured facts from a graph composed of entities and their relationships (triples: subject–predicate–object).
Advantages:
- Rich **semantic structure** allows reasoning and inferencing.
- High interpretability and **useful in high-precision domains** like medicine or finance.
- Enables **logical and multi-hop queries**.

Disadvantages:
- **Lower retrieval efficiency** due to graph traversal or semantic query matching.
- Quality and coverage **highly dependent on KG construction**.
- May **miss nuanced or unstructured context**, limiting usefulness in natural language tasks.

When to retrieve?

Single Search (Pre-retrieval): Retrieve once before generation begins, typically based on the full user query or prompt.
Advantages:
- **Low latency**: Retrieval occurs only once.
- Simpler to implement and easier to scale.
- Encourages consistency throughout the generated output.

Disadvantages:
- Cannot adjust to new emerging needs mid-generation.
- Prone to **context mismatch** if the initial query is ambiguous or multi-turn.

Best For: Factual QA, summarization, and when initial intent is clear.

Retrieval Strategy	Reactivity	Latency	Cost	Context Relevance	Best Use Cases
Single Search	Low	Low	Low	Medium	Clear factual QA, summarization
Each Token	Very High	Very High	Very High	High	Research, exploratory dialogue
Every N Tokens	Medium	Medium	Medium	Medium–High	Long-form generation, storytelling
Adaptive Search	High	Low–High	Variable	High	Assistants, dynamic QA, complex generation

Each Token (Token-Level Retrieval): Retrieve context at **every generation step** (i.e., for each output token).
Advantages:
- **Highly dynamic and reactive**—maximizes adaptability.
- Supports fine-grained alignment between context and generated content.

Disadvantages:
- **Extremely expensive computationally**—very high latency.
- May lead to inconsistency or instability in generation if context changes rapidly.

Best For: Research prototypes, highly adaptive systems, and dynamic generation tasks where context changes frequently (e.g., live dialogue with external sources).

Every N Tokens: Retrieve context **periodically**, for example every 20, 50, or 100 tokens.
Advantages:
- **Balance between responsiveness and efficiency**.
- Allows system to adapt to topic drift or evolving user needs.
- Can be tuned based on task complexity or token budget.

Disadvantages:
- May still miss context shifts between intervals.
- Requires careful tuning to avoid stale or irrelevant context.

Best For: Long-form generation, summarization, open-ended responses, and creative writing where the topic evolves gradually.

Adaptive Search (Conditional Retrieval): Trigger retrieval **dynamically based on signals**, such as a) Detection of uncertainty (e.g., high entropy in the model's predictions); b) User-driven clarification; c) Detection of entities or topics needing enrichment.
Advantages:
- **Smart, efficient use of retrieval resources**.
- Context is injected **only when needed**, optimizing cost and relevance.
- Offers better **human-like reasoning** (e.g., "let me check that").

Disadvantages:
- **Complex implementation**—requires monitoring model state or uncertainty.
- Needs fine-tuned thresholds or triggers to avoid over/under-retrieval.

Best For: Advanced assistant systems, decision support, multimodal interactions, exploratory dialogue.

How to use the retrieved information?

Layer	Integration Depth	Model Change Required	Pros	Cons	Best Use Cases
Input/ Data Layer	Shallow	No	Simple, scalable, fast	Limited semantic alignment, token limit	QA, retrieval-augmented prompting
Model Layer	Deep	Yes	High semantic integration, reasoning	Complex to build, expensive	Multi-hop QA, knowledge-intensive tasks
Output Layer	Post hoc	No	Improves reliability, verifiability	No direct influence on generation process	Post-processing, ranking, self-correction

Input/Data Layer (Prepend or Concatenate Retrieval): Retrieved content is appended to the input prompt before feeding it into the language model.
Advantages:
- **Simplicity**: Most widely used in production (e.g., OpenAI's function calling, LangChain).
- Works with **frozen models** (no architectural changes needed).
- Easy to implement and debug.

Disadvantages:
- **Token budget limits**: Large retrieved content can exceed model input limits.
- No deep integration—model may **ignore or underuse** retrieved context.
- Hard to differentiate between important and background info.

Best For: Document QA, summarization, factual recall, early RAG implementations.

Model/Intermediate Layer (Cross-Attention, Fusion-in-Decoder): Injects retrieved content **inside the model's architecture**, typically through: a) **Cross-attention heads** (e.g., FiD—Fusion-in-Decoder); b) **Memory mechanisms** or external knowledge encoders.
Advantages:
- Deep semantic **integration of retrieval and generation**.
- Allows **multi-document reasoning** and selective focus on retrieved info.
- More **robust to noise** in retrieved content.

Disadvantages:
- Requires **model modification or fine-tuning**.
- More complex and **harder to scale or deploy**.
- Computational cost increases significantly.

Best For: Multi-hop QA, reasoning tasks, and when high fidelity to retrieved data is essential.

Output/Prediction Layer (Post-Generation Conditioning or Rewriting): Retrieved content is used **after generation**, to: a) **Guide selection** among generated outputs; b) **Filter, validate, or rerank** answers; c) Support **fact-checking, grounding, or explanation generation**.
Variants:
- Score-and-rank multiple generated hypotheses.
- Retrieve after initial generation and rephrase with retrieved data.

Advantages:
- Enhances **output reliability** via post hoc validation.
- No need to modify generation architecture.
- Can support **self-refinement or correction** loops.

Disadvantages:
- Less integrated—retrieval does not shape generation directly.
- May lead to **incoherent edits or inconsistencies** if used naively.

Best For: Assistant systems, generative QA pipelines with verifiability, or LLMs that self-reflect.

Issues of RAG

Issues of RAG

Augmentation stage

- Pre-training
- Fine-tuning
- Inference

Retrieval choice

- BERT
- Roberta
- BGE
- ...

Generation choice

- GPT
- Llama
- T5
- ...

Issues of RAG

Augmentation stage

Augmentation Stage Issues
- **Mismatched retrieval and generation**: Retrieved documents may not align well with the needs of the generator
- **Noisy context**: Irrelevant or overly verbose chunks can confuse the generation model
- **Lack of grounding**: Model may hallucinate even when relevant content is provided

Pre-training Issues
- **Domain mismatch**: Pretrained models may not understand domain-specific terms or logic
- **Limited context awareness**: Models trained on general data may struggle with nuanced reasoning
- **Inconsistent tokenization**: Differences between retrieval and generation model tokenizers can hurt performance

Fine-Tuning Issues
- **Overfitting**: Fine-tuning on small datasets can lead to brittle, overly specific behavior
- **Catastrophic forgetting**: Model may lose general capabilities during fine-tuning
- **Data quality**: Low-quality fine-tuning data leads to unreliable or biased outputs

Inference-Time Issues
- **Latency**: Real-time retrieval + generation can be slow without optimization
- **Prompt construction**: Poor prompt engineering can lead to subpar outputs
- **Context limit**: LLMs may not handle long retrieved texts well due to token limits

Retrieval choice

Model	Common Issues
BERT	Shallow understanding of full context, limited scalability
RoBERTa	Similar to BERT but larger; still limited by bi-encoder constraints
BGE	May require careful fine-tuning for specific tasks/domains
(Others)	Quality varies based on pretraining data and embedding granularity

Generation choice

Model	Common Issues
GPT	Prone to hallucination without strong grounding, expensive inference
LLaMA	May underperform without fine-tuning, especially on open-ended tasks
T5	Designed for text-to-text but may struggle with long context
(Others)	Trade-offs in cost, fluency, controllability, and token limits

Evaluation of RAG

Evaluate the Effectiveness of RAG

Retrieval Evaluation

- See if the system finds the **right information**
- Check **how close the results are to what's needed**
- Use **test data** or **review results manually** to see if they're helpful

Generation Evaluation

- Make sure the answer is **based on real facts, not made up**
- See if the answer **sounds clear** and **makes sense**
- Compare answers to correct ones to **check accuracy**

End-to-End Evaluation

- Test if the **system gives the right answer overall**
- **Ask users** if they're happy with the results
- Measure **how fast it responds**

Holistic Analysis & Tools

- Use tools like **RAGAS** or **LangChain** to test your system easily
- Let an **AI model help judge the quality of answers**
- **Check where problems happen** — in retrieval, generation, or both

Evaluate the Effectiveness of RAG

Retrieval Evaluation

Determine how well the system retrieves relevant documents/chunks.

Key Metrics

- **Recall@k**: Proportion of relevant documents retrieved in the top *k* results
- **Precision@k**: Proportion of retrieved documents in the top *k* that are relevant
- **Mean Reciprocal Rank (MRR)**: Measures how soon the first relevant document appears
- **Hit Rate**: Binary metric indicating whether at least one relevant document was retrieved
- **Embedding similarity scores**: Average cosine similarity between query and top retrieved chunks

How to Test

- Use a benchmark dataset with labeled query–document pairs (e.g., Natural Questions, FiQA)
- Manually annotate retrieved results for relevance if no dataset is available

Generation Evaluation

Assess the quality, accuracy, and relevance of generated answers.

Key Metrics

- **Factual Accuracy**: Whether the generated output is grounded in the retrieved content
- **Faithfulness**: Absence of hallucinations or unsupported claims
- **Relevance**: How well the answer matches the user's query
- **Fluency/Coherence**: Naturalness and readability of the output
- **Conciseness**: Avoiding verbosity while preserving meaning

Automated Metrics

- **ROUGE**: Measures text overlap with reference answers (useful for summarization-like tasks)
- **BLEU**: Useful for structured/short outputs, but less so for long-form text
- **BERTScore**: Uses contextual embeddings to compare similarity between generated and reference texts
- **Faithfulness-specific metrics**: Like **FactCC**, **QAGS**, or **FEVER scores** (for factual consistency)

End-to-End Evaluation

Measure how well the entire RAG pipeline performs on real tasks.

User-Level or Task-Specific Metrics

- **Answer Accuracy**: % of questions answered correctly in a QA task
- **Exact Match (EM)**: Checks if the answer exactly matches a gold answer
- **Human Evaluation**: Subjective scoring on helpfulness, informativeness, and correctness
- **Latency**: Time taken to generate responses
- **User Satisfaction Score (UX surveys, A/B tests)**

Holistic Analysis & Tools

Useful Tools:

- **RAGAS**: Framework to evaluate RAG using built-in retrieval and generation metrics
- **LangChain Evaluation Toolkit**: For evaluating retrieval, prompt quality, and outputs
- **LLM-as-a-Judge**: Use a trusted LLM to score the relevance, faithfulness, and helpfulness of outputs

Diagnostic Checks:

- Analyze when **retrieval fails** vs when **generation hallucinates**
- Visualize **embedding space** to inspect clustering of similar queries
- Examine **retrieval coverage**: How much of the final answer was supported by retrieved content

Python Code Walkthrough

RAG Execution Preparation

RAG PDF Chatbot with OpenRouter API

- **GitHub:** https://github.com/deepansarkar/rag-basic
- **OpenRouter:** https://openrouter.ai/
- **OpenRouter DeepSeek LLM:** https://openrouter.ai/deepseek/deepseek-r1-0528-qwen3-8b:free
- **OpenRouter Keys:** https://openrouter.ai/settings/keys

PDF Documents required for RAG to work

- Some samples are kept in **sample_data** folder

Environment Variables in .env file

```
OPENROUTER_API_KEY=Create From <https://openrouter.ai/settings/keys>
OPENROUTER_API_URL=https://openrouter.ai/api/v1/chat/completions
MIN_TRIES=5
LLM_MODEL=deepseek/deepseek-r1-0528:free
```

Main Function to Start Execution

Initialize the RAGChat instance which loads and processes all PDFs at startup

Start an infinite loop to interact with the user

If the user entered a non-empty question
- Pass the question to the RAG pipeline and get the answer
- Print the answer as a raw multiline response
- Print a blank line for visual spacing

```python
from src.rag_pipeline import RAGChat

def main():
    print("📄 RAG PDF Chat")
    rag = RAGChat()
    print("Type your question or type 'exit' or 'quit' to close the conversation.\n")

    while True:
        try:
            question = input("🧠 You: ").strip()
            if question.lower() in {"exit", "quit"}:
                print("👋 Exiting. Goodbye!")
                break
            if question:
                answer = rag.ask(question)
                print("🤖 Answer:\n")
                print(answer)
                print("")
            else:
                print("⚠️ Please enter a question.")
        except KeyboardInterrupt:
            print("\n👋 Exiting. Goodbye!")
            break
        except Exception as e:
            print(f"❌ Error: {e}")

if __name__ == "__main__":
    main()
```

Display a title banner for the chat interface

Prompt the user for input with usage instructions

Prompt user for input and strip whitespace

If user types "exit" or "quit", terminate the program gracefully

If the user submitted an empty question

Handle Ctrl+C gracefully by exiting the loop

Catch and display any other unexpected errors

Basic RAG Pipeline

Class to manage Retrieval-Augmented Generation (RAG) over PDF documents
- Set the folder path where PDFs are located
- Initialize the vector store for embedding and retrieval
- Load and process all PDFs in the folder into chunks and embeddings

Called once during initialization to process all PDFs to cache chunks and their embeddings for all files

Loop through each file in the specified PDF folder

- Add the current file's chunks to the global list
- Append the file's embeddings to the embedding list

Answers a question using the cached chunks and embeddings
- Retrieve the top-k most relevant chunks based on query similarity
- Join the top chunks into a single context string for the model
- Query the OpenRouter model with the question and extracted context

```python
import os
from src.pdf_loader import load_pdf, chunk_text_simple
from src.vector_store import VectorStore
from src.openrouter_api import query_openrouter
from torch import cat

class RAGChat:
    def __init__(self, pdf_folder="data/pdf"):
        self.pdf_folder = pdf_folder
        self.vstore = VectorStore()
        self.chunks, self.embeddings = self._load_all_pdfs()

    def _load_all_pdfs(self):
        all_chunks = []
        all_embeddings = []
        for file in os.listdir(self.pdf_folder):
            if file.lower().endswith(".pdf"):
                path = os.path.join(self.pdf_folder, file)
                raw_text = load_pdf(path)
                chunks = chunk_text_simple(raw_text)
                chunks, embeddings = self.vstore.load_or_create(file, chunks)
                all_chunks.extend(chunks)
                all_embeddings.append(embeddings)
        if not all_chunks:
            raise ValueError("No valid PDFs found in data/pdfs.")
        combined_embeddings = cat(all_embeddings, dim=0)
        return all_chunks, combined_embeddings

    def ask(self, question: str, top_k=3) -> str:
        top_chunks = self.vstore.retrieve_top_k(self.chunks, self.embeddings, question, k=top_k)
        context = "\n\n".join(top_chunks)
        return query_openrouter(question, context)
```

Initialize empty lists to hold all chunks and embeddings

- Load and extract raw text from the PDF
- Split the raw text into smaller simple chunks
- Load cached embeddings or compute and cache them if missing

If no valid PDF content was found, raise an error

- Concatenate all individual tensors into one combined tensor for efficient querying
- Return the combined chunks and embeddings

Open Router API

Load environment variables from a .env file into the system environment

```python
import os
import requests
import time
from dotenv import load_dotenv

load_dotenv()

API_KEY = os.getenv("OPENROUTER_API_KEY")
API_URL = os.getenv("OPENROUTER_API_URL")
MIN_TRIES = int(os.getenv("MIN_TRIES"))
LLM_MODEL = os.getenv("LLM_MODEL")

headers = {
    "Authorization": f"Bearer {API_KEY}",
    "Content-Type": "application/json",
}
```

Define the headers required for the API request

- Bearer token for API authentication
- Specify content type as JSON

Retrieve required configuration values from the environment

- API key for OpenRouter authentication
- URL endpoint for the OpenRouter API
- Number of retry attempts
- Language model to use

Calls the OpenRouter API with a question and context to retrieve an answer. If the API call fails, it retries multiple times (up to MIN_TRIES)

```python
def query_openrouter(question, context):
    prompt = f"""Use the context below to answer the question. If the answer isn't in the context, say "I am not able to answer based on provided context".

    Context:
    {context}

    Question: {question}
    """

    data = {
        "model": LLM_MODEL,
        "messages": [
            {
                "role": "system",
                "content": "You are a helpful assistant who answers question based on context from provided data. Reply with concise and accurate information in a single paragraph."
            },
            {
                "role": "user",
                "content": prompt
            }
        ]
    }

    for attempt in range(1, MIN_TRIES + 1):
        try:
            response = requests.post(API_URL, headers=headers, json=data, timeout=15)
            if response.status_code == 200:
                return response.json()["choices"][0]["message"]["content"].strip()
            else:
                print(f"Attempt {attempt}: Status {response.status_code}")
        except Exception as e:
            print(f"Attempt {attempt}: Exception - {e}")
        time.sleep(1)

    return "Failed to get a valid response after multiple attempts."
```

Create the prompt that guides the model using a Retrieval-Augmented Generation (RAG) style

Construct the data payload to send to the OpenRouter API

If the request was successful (HTTP 200 OK), parse and return the model's response, else log the HTTP status code for non-successful attempts

Retry loop: attempt the request up to MIN_TRIES times

Make the POST request to the OpenRouter API

Log any exceptions that occur during the request

Wait 1 second before retrying to avoid rapid repeated requests

If all attempts fail, return an error message

PDF Loader

- Create a PdfReader object to access the PDF
- Initialize an empty string to store the cumulative text

Loads a PDF file and extracts all text from each page

Loop through all pages in the PDF

- Extract text from the current page
- Append extracted text to the cumulative text if it exists
- Add a newline after each page's content

Return the complete extracted text

```python
from PyPDF2 import PdfReader

def load_pdf(file_path: str) -> str:
    reader = PdfReader(file_path)
    text = ""
    for page in reader.pages:
        page_text = page.extract_text()
        if page_text:
            text += page_text + "\n"
    return text
```

Splits a long string of text into smaller chunks of fixed size

The number of characters per chunk has to be set (default is 500)

List comprehension that iterates through the text in steps of `chunk_size`, extracting and trimming each chunk

```python
def chunk_text_simple(text: str, chunk_size: int = 500) -> list:
    return [text[i:i+chunk_size].strip() for i in range(0, len(text), chunk_size)]
```

Vector Store

Initializes the VectorStore with a SentenceTransformer model and a cache directory

- Load a pre-trained sentence embedding model (small and fast)
- Set the directory where cached embeddings will be stored (defaults to 'data/cache')
- Create the cache directory if it does not already exist

```python
import os
import pickle
from sentence_transformers import SentenceTransformer, util

class VectorStore:
    def __init__(self, cache_dir="data/cache"):
        self.model = SentenceTransformer("all-MiniLM-L6-v2")
        self.cache_dir = cache_dir
        os.makedirs(cache_dir, exist_ok=True)

    def load_or_create(self, pdf_name: str, chunks: list):
        base = os.path.splitext(pdf_name)[0]
        cache_file = os.path.join(self.cache_dir, base + ".pkl")
        if os.path.exists(cache_file):
            with open(cache_file, "rb") as f:
                print(f"Loaded cached data for {pdf_name}")
                return pickle.load(f)
        embeddings = self.model.encode(chunks, convert_to_tensor=True)
        with open(cache_file, "wb") as f:
            pickle.dump((chunks, embeddings), f)
            print(f"Cached embeddings for {pdf_name}")
        return chunks, embeddings

    def retrieve_top_k(self, chunks, embeddings, query, k=3):
        query_embedding = self.model.encode(query, convert_to_tensor=True)
        scores = util.cos_sim(query_embedding, embeddings)[0]
        top_indices = scores.topk(k)[1]
        return [chunks[i] for i in top_indices]
```

Loads cached embeddings for a given PDF if available; otherwise computes and caches them

- Derive the base name of the PDF (without extension)
- Create full path to the cache file using the base name

- Check if cached file exists
- Open the cache file and load the contents

Compute embeddings for each text chunk using the model

Save both the chunks and their embeddings into a binary cache file

Return the chunks and computed embeddings

Finds the top-k most relevant text chunks for a given query using cosine similarity

Number of top similar chunks to retrieve (defaults to 3)

Encode the query into an embedding vector

Compute cosine similarity between query embedding and all chunk embeddings

Retrieve the indices of the top-k most similar scores

Return the top-k matching chunks based on the retrieved indices

Thank You