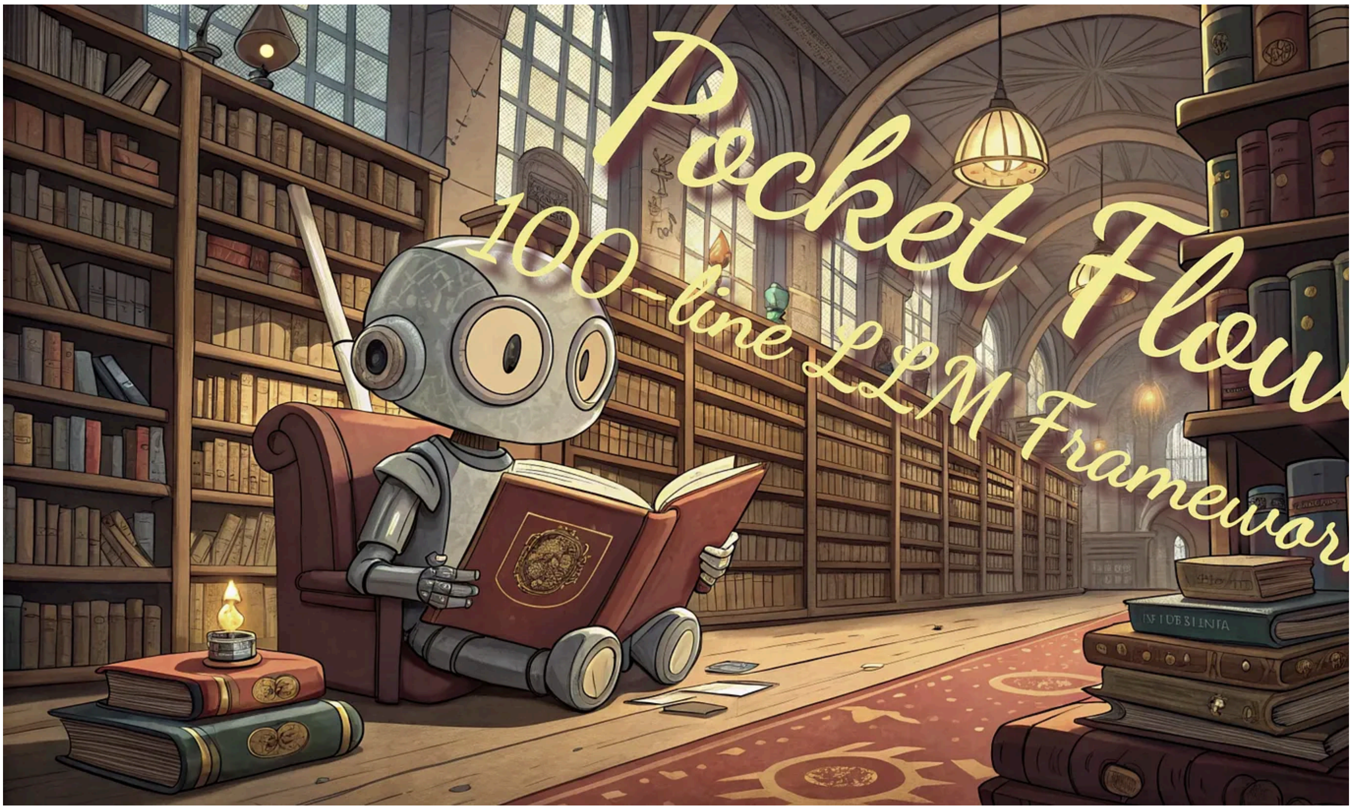
Retrieval Augmented Generation (RAG) from Scratch — Tutorial For Dummies

ZACHARY HUANG
APR 01, 2025



> *Ever wondered how AI tools like ChatGPT can answer questions based on specific documents they've never seen before? This guide breaks down Retrieval Augmented Generation (RAG) in the simplest possible way with [minimal code implementation](#)!*

Have you ever asked an AI a question about your personal documents and receivec
completely made-up ro
outdated informatior ge
Language Models), b AG

In this beginner-frie

- The key concepts behind RAG systems in plain language
- How to build a working RAG system step-by-step
- Why RAG dramatically improves AI responses for your specific documents

---

We'll use [PocketFlow](#) - a simple 100-line framework that strips away complexity. Unlike frameworks with convoluted abstractions, PocketFlow lets you see the entire system at o giving you the fundamentals to build your understanding from the ground up.

What's RAG (In Human Terms)?

Imagine RAG is like giving an AI its own personal research librarian before it ans your questions. Here's how the magic happens:

1. **Document Collection**: You provide your documents (company manuals, articl books) to the system, just like books being added to a library.
2. **Chunking**: The system breaks these down into bite-sized, digestible pieces - l librarians dividing books into chapters and sections rather than working with entire volumes.
3. **Embedding**: Each chunk gets converted into a special numerical format (vect that captures its meaning - similar to creating detailed index cards that understand concepts, not just keywords.
4. **Indexing**: These ica card catalog that
5. **Retrieval**: When e r relevant chunks

6. **Generation**: The AI crafts an answer using both your question AND these hel references, producing a much better response than if it relied solely on its pre trained knowledge.

The result? Instead of making things up or giving outdated information, the AI grounds its answers in your specific documents, providing accurate, relevant responses tailored to your information.

Chunking: Breaking Documents into Manageabl Pieces

Before our RAG system can work effectively, we need to break our documents intc smaller, digestible pieces. Think of chunking like preparing a meal for guests - you wouldn't serve a whole turkey without carving it first!

Why Chunking Matters

The size of your chunks directly impacts the quality of your RAG system:

- **Too large chunks**: Your system retrieves too much irrelevant information (like serving entire turkeys)
- **Too small chunks**: You lose important context (like serving single peas)
- **Just right chunks**: Your system finds precisely what it needs (perfect portions

Let's explore some practical chunking methods:

1. Fixed-Size Chunking: Simple but Imperfect

The simplest approach divides text into equal-sized pieces, regardless of content:

```
def fixed_size_ch
    chunks = []
    for i in rang
        chunks.ap
    return chunks
```

This code loops through text, taking 50 characters at a time. Let's see how it work a sample paragraph:

Input Text:

```
The quick brown fox jumps over the lazy dog. Artificial intelligence h
revolutionized many industries. Today's weather is sunny with a chance
of rain. Many researchers work on RAG systems to improve information
retrieval.
```

Output Chunks:

```
Chunk 1: "The quick brown fox jumps over the lazy dog. Arti"
Chunk 2: "ficial intelligence has revolutionized many indus"
Chunk 3: "tries. Today's weather is sunny with a chance of "
Chunk 4: "rain. Many researchers work on RAG systems to imp"
Chunk 5: "rove information retrieval."
```

Notice the problem? The word "Artificial" is split between chunks 1 and 2. "Industries" is split between chunks 2 and 3. This makes it hard for our system to understand the content properly.

2. Sentence-Based Chunking: Respecting Natural Boundaries

A smarter approach is to chunk by complete sentences:

```
import nltk  # Natural Language Toolkit library

def sentence_based chunk(text, max sentences=1)
    sentences = r
    chunks = []
    # Group sente
    for i in rang
        chunks.ap
    return chunks
```

This method first identifies complete sentences, then groups them one at a time:

Output Chunks:

```
Chunk 1: "The quick brown fox jumps over the lazy dog."
Chunk 2: "Artificial intelligence has revolutionized many industries.'
Chunk 3: "Today's weather is sunny with a chance of rain."
Chunk 4: "Many researchers work on RAG systems to improve information
retrieval."
```

Much better! Each chunk now contains a complete sentence with its full meaning intact.

3. Additional Chunking Strategies

Depending on your documents, these approaches might also work well:

- **Paragraph-Based**: Split text at paragraph breaks (usually marked by newlines)
- **Semantic Chunking**: Group text by topics or meaning (often requires AI assistance)
- **Hybrid Approaches**: Combine multiple strategies for optimal results

Choosing the Right Approach

While many sophisticated chunking methods exist, for most practical applications "Keep It Simple, Stupid" (KISS) principle applies. **Starting with Fixed-Size Chunk of around 1,000 characters per chunk is often sufficient** and avoids overcomplica your system.

The best chunking app *ıse*
like

Embeddings: Making Retrieval Possible

Now that we've chopped up our documents, how does the system find the most relevant chunks for our questions? This is where **embeddings** come in! Embeddin are the magic that powers our retrieval system. Without them, we couldn't find th right information for our questions!

What Are Embeddings?

An embedding transforms text into a list of numbers (a vector) that captures its meaning. Think of it as creating a special "location" in a meaning-space where sir ideas are positioned close together.

For example, if we take these three sentences:

1. "The cat sat on the mat." 🐱
2. "A feline rested on the floor covering." 🐈
3. "Python is a popular programming language." 💻

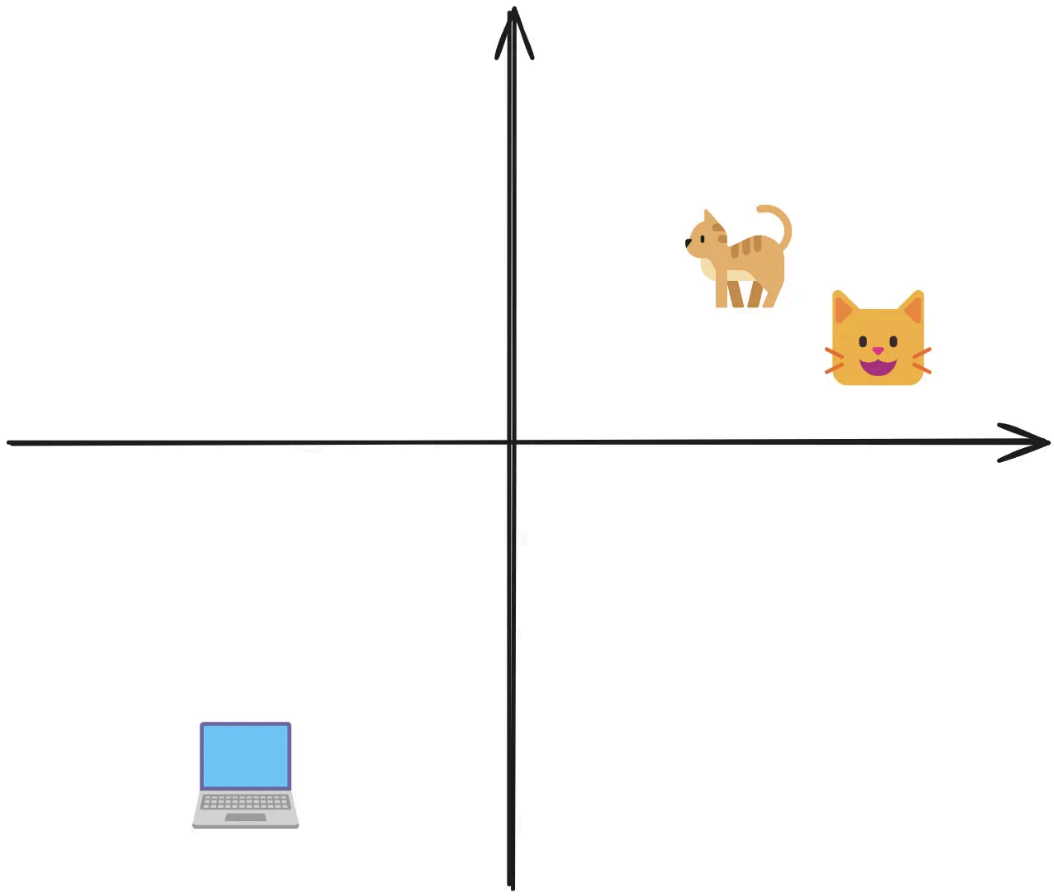
A good embedding system would place sentences 1 and 2 close together (since the describe the same concept), while sentence 3 would be far away (completely differe topic).

Visualizing Embeddings in Action

When we convert these sentences to embeddings, we might get something like thi a simplified 2D space:

```
Sentence 1 (🐱): [0.8, 0.2]
Sentence 2 (🐈):
Sentence 3 (💻):
```


On a graph, the cat-r e programming senten nti relationships!



With embeddings, measuring similarity is as simple as calculating the distance between points:

- Distance from 🐱 to 🐈: Very small (about 0.14 units) → Very similar!
- Distance from 🐱 to 💻: Very large (about 1.95 units) → Not related at all!

This confirms what we intuitively know - the cat sentences are closely related, wh the programming sentence is completely different.

Creating Embeddings: From Simple to Advanced

1. Simple Approach: Character Frequency

Here's a beginner-fri


```
def get_simple_er
    # Create a ve
    embedding = r
```

```
    # Count character frequencies in the text
    for char in text:
        embedding[ord(char) % 26] += 1.0

    # Normalize the vector
    norm = np.linalg.norm(embedding)
    if norm > 0:
        embedding /= norm

    return embedding
```

Let's see what this looks like for our three example sentences:

```
🐱 "The cat sat on the mat." → [0.2, 0, 0, ...]
(high values at positions for 't', 'a', ' ', etc.)

🐈 "A feline rested on the floor covering." → [0.1, 0.2, 0, ...]
(different characters but similar number of spaces)

💻 "Python is a popular programming language." → [0.1, 0.2, 0.1, ...]
(completely different character pattern)
```

Notice how sentences 1 and 2 have different values because they use different wor while sentence 3 has a very different pattern altogether. This simple vector only captures the characters used, not their meaning.

Limitation: This simple approach can't recognize that "cat" and "feline" are relate concepts since they share no characters!

2. Professional Approach: AI-Powered Embeddings

In a real RAG system, you'd use sophisticated models like OpenAI's embedding A


```
def get_openai_er
    client = Oper
"YOUR_API_KEY"))

    response = client.embeddings.create(
```

```
        model="text-embedding-ada-002",
        input=text
    )

    # Extract the embedding vector from the response
    embedding = response.data[0].embedding

    return np.array(embedding, dtype=np.float32)
```

These advanced embeddings capture deep semantic relationships in high-dimensi space (typically 1,536 dimensions for OpenAI's embeddings). They understand tha "cat" 🐱 and "feline" 🐈 mean the same thing, even with completely different characters!

```
🐱 "The cat sat on the mat." → [0.213, -0.017, 0.122, ...]
(pattern encoding "animal resting on object")

🐈 "A feline rested on the floor covering." → [0.207, -0.024, 0.118,
...]
(very similar pattern, also encoding "animal resting on object")

💻 "Python is a popular programming language." → [-0.412, 0.158, 0.36
...]
(completely different pattern encoding "programming language")
```

Even though sentences 1 and 2 use different words, their embeddings have similar patterns because they express similar concepts. Meanwhile, sentence 3's embeddi has a completely different pattern because it's about a different topic entirely.

How Embeddings Drive Our RAG System

In our RAG pipeline:

1. We embed all do
2. When a user ask

3. We find the document chunks whose embeddings are closest to the question's embedding

---

However, as your document collection grows into thousands or millions of chunks, searc through all embeddings becomes slow. That's where vector databases come in - speciali systems designed to make this search lightning-fast!

Vector Databases: Making Retrieval Fast

Imagine having to search through a million book pages to find an answer - it woul take forever! Vector databases solve this problem by organizing our embeddings in clever way that makes searching lightning-fast.

Why We Need Vector Databases

When you ask a question, your RAG system needs to compare it against potentiall thousands or millions of document chunks. There are two ways to do this:

1. Simple Approach: The Exhaustive Search

In this approach, we check every single document chunk one by one - like going through every page in a library:

```
def retrieve_naive(question_embedding, chunk_embeddings):
    best_similarity, best_chunk_index = -1, -1

    for idx, chunk_embedding in enumerate(chunk_embeddings):
        similarity = get_similarity(question_embedding, chunk_embeddin
        if simila
            best_

    return best_c
```

This works perfectly for a few dozen documents, but becomes painfully slow with thousands or millions of chunks.

2. Professional Approach: Smart Indexing with Vector Databases

Vector databases are like having a magical librarian who organizes books so effect that they can instantly find what you need without checking every shelf.

Let's see how this works in two simple steps:

Step 1: Building the Magical Index

First, we organize all our document embeddings into a special structure:

```
def create_index(chunk_embeddings):
    dimension = chunk_embeddings.shape[1]  # e.g. 128 or 1536
    index = faiss.IndexFlatL2(dimension)   # flat = exact search
    index.add(chunk_embeddings)            # add all document vectors
    return index
```

This is like creating a detailed map of where every document "lives" in our meanin space.

Step 2: Fast Retrieval Using the Index

When a question comes in, we use our magical index to find the most relevant chu in an instant:

```
def retrieve_index(index, question_embedding, top_k=5):
    _, chunk_indices = index.search(question_embedding, top_k)
    return chunk_indices
```

Instead of checking e vir the top 5 most releva

What Makes V

The speed of vector databases comes from three clever techniques:

1. **Smart Indexing Algorithms**: Methods like **HNSW** (Hierarchical Navigable Sn Worlds) create shortcuts through the embedding space, so the system only ne to check a small fraction of documents.
2. **Vector Compression**: These databases can shrink embeddings to save memor while preserving their relationships - like having a compressed map that still shows all the important landmarks.
3. **Parallel Processing**: Modern vector databases use multiple CPU/GPU cores simultaneously, checking many possibilities at once - like having a team of librarians all searching different sections of the library at the same time.

With these techniques, vector databases can search through millions of documents ir milliseconds - making RAG systems practical for even the largest document collection

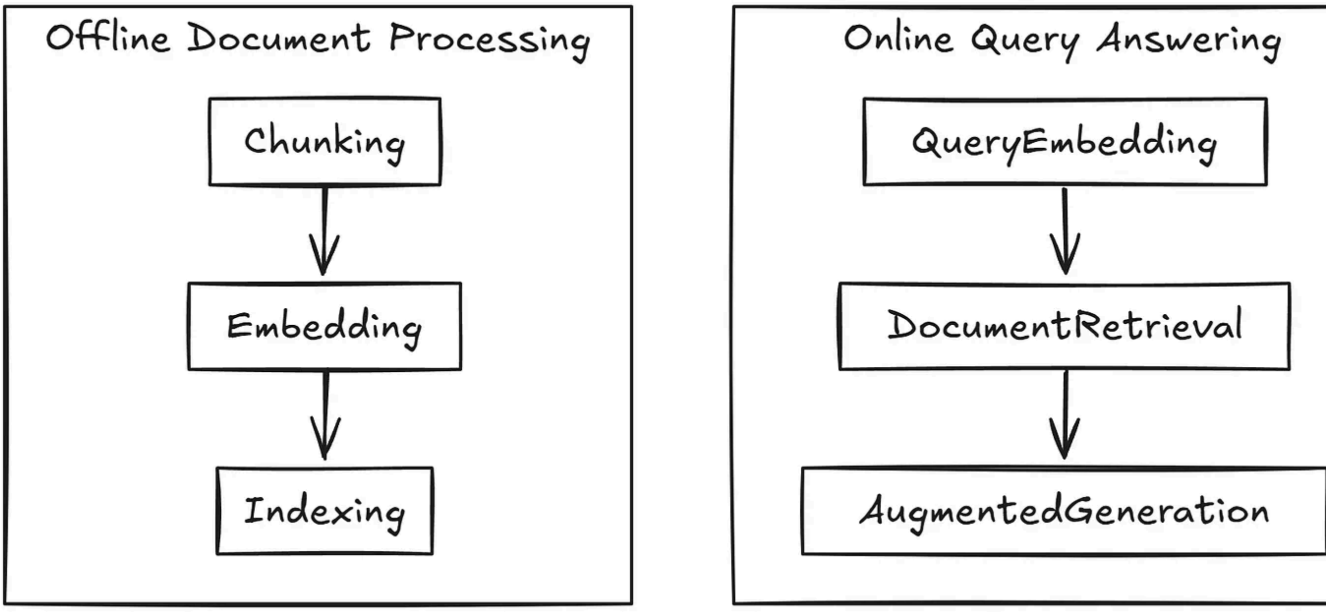
---

Putting RAG Together: Just Two Simple Workflo

Now that you understand chunking, embeddings, and vector databases, here's the beautiful simplicity of RAG that many frameworks overcomplicate:

> *RAG is just two straightforward workflows working together!*

Think of RAG like your personal research assistant, organized into two efficient processes:



1. Offline Flow: Preparing Your Knowledge Base

This happens just once, before any questions are asked:

- **Chunking**: Break documents into bite-sized pieces (like dividing a book into memorable paragraphs)
- **Embedding**: Convert each chunk into a numerical vector (like giving each paragraph a unique "location" in meaning-space)
- **Indexing**: Organize these vectors for efficient retrieval (like creating a magica map of all your knowledge)

2. Online Flow: Answering Questions in Real-Time

This happens each time someone asks a question:

- **Query Embedding**: Transform the question into a vector (finding its "location the same meaning-space)
- **Retrieval**: Find t ... st relevant knowle
- **Augmented Gen ... rie context (craft a r

That's it! The magic of RAG isn't in complex algorithms - it's in how these two simpl workflows work together to provide accurate, relevant answers based on your specific knowledge!

Building RAG with PocketFlow

PocketFlow makes building RAG systems delightfully simple. Unlike complex frameworks with layers of abstraction, PocketFlow uses minimal building blocks i just 100 lines of code to clearly show how everything works.

Think of PocketFlow as a well-organized kitchen where:

- **Nodes** are cooking stations that perform specific tasks:

```
class BaseNode:
    def __init__(self): self.params, self.successors = {}, {}
    def add_successor(self, node, action="default"):
        self.successors[action] = node
        return node
    def prep(self, shared): pass
    def exec(self, prep_res): pass
    def post(self, shared, prep_res, exec_res): pass

    def run(self, shared):
        p = self.prep(shared);
        e = self.exec(p);
        return self.post(shared, p, e)
```

- **Flow** is the recip


```
class Flow(BaseN
    def __init__(                                                          rt
    def get_next_
        return curr.successors.get(action or "default")
```

```
    def orch(self, shared, params=None):
        curr, p = copy.copy(self.start), (params or {**self.params})
        while curr:
            curr.set_params(p)
            c = curr.run(shared)
            curr = copy.copy(self.get_next_node(curr, c))

    def run(self, shared):
        pr = self.prep(shared)
        self.orch(shared)
        return self.post(shared, pr, None)
```

- **Shared Store** is a common workspace where all nodes can access ingredients:

```
# Connect nodes together
load_data_node = LoadDataNode()
summarize_node = SummarizeNode()
load_data_node >> summarize_node

# Create flow
flow = Flow(start=load_data_node)

# Pass data through shared store
shared = {"file_name": "data.txt"}
flow.run(shared)
```

Each **Node** performs three simple operations:

- **Prep**: Gather what's needed from the shared store
- **Exec**: Perform its specific task
- **Post**: Store resul

The **Flow** manages tl ... :he
next based on specifi

With these simple building blocks, PocketFlow makes RAG systems easy to build, unders and modify!

Building a RAG From Scratch with PocketFlow

Let's create an AI assistant that can answer questions from your documents. Our is to build a simple but powerful RAG system that:

1. Processes your documents only once (offline flow)
2. Takes user questions
3. Retrieves the most relevant information
4. Generates accurate, document-grounded answers

Step-By-Step Walkthrough with a Real Example

Let's see what happens when you ask our system:

> *What's the capital of France?*

Offline Phase (Happens Once)

1. **ChunkDocuments Node**:
    - **Input** (`Prep`): Your collection of documents
    - **Process** (`Exec`): Divides each document into manageable chunks
    - **Output** (`Post`): Collection of document chunks (including one with "Paris the capital of France")
2. **EmbedDocumen
    - **Input** (`Prep`
    - **Process** (`Exe`
    - **Output** (`Post`): Collection of document embeddings

3. **CreateIndex Node**:
    - **Input** (`Prep`): Document embeddings
    - **Process** (`Exec`): Builds a searchable vector index
    - **Output** (`Post`): Ready-to-use vector database

Online Phase (Happens Every Question)

1. **EmbedQuery Node**:
    - **Input** (`Prep`): "What's the capital of France?"
    - **Process** (`Exec`): Converts question into an embedding vector
    - **Output** (`Post`): Question embedding
2. **RetrieveDocuments Node**:
    - **Input** (`Prep`): Question embedding + vector database
    - **Process** (`Exec`): Finds the most similar document chunk
    - **Output** (`Post`): Retrieves "Paris is the capital of France" chunk
3. **GenerateAnswer Node**:
    - **Input** (`Prep`): Original question + retrieved chunk
    - **Process** (`Exec`): Crafts response using both inputs
    - **Output** (`Post`): "The capital of France is Paris."

Now let's implement this with code!

Building the Offline Flow

Node 1: The ChunkDocumentsNode

This node breaks doc


```
class ChunkDocume
    def prep(sel
        return shared["texts"]
```

```
    def exec(self, text):
        return fixed_size_chunk(text)

    def post(self, shared, prep_res, exec_res_list):
        all_chunks = []
        for chunks in exec_res_list:
            all_chunks.extend(chunks)

        shared["texts"] = all_chunks
        print(f"✅ Created {len(all_chunks)} chunks")
        return "default"
```

The `prep` gets documents from shared storage, `exec` divides each document into chunks, and `post` combines all chunks into a single list for the next node.

Node 2: The EmbedDocumentsNode

This node converts each text chunk into a numerical vector:

```
class EmbedDocumentsNode(BatchNode):
    def prep(self, shared):
        return shared["texts"]

    def exec(self, text):
        return get_embedding(text)

    def post(self, shared, prep_res, exec_res_list):
        embeddings = np.array(exec_res_list, dtype=np.float32)
        shared["embeddings"] = embeddings
        print(f"✅ Created {len(embeddings)} document embeddings")
        return "default"
```

The `prep` retrieves t vec

and `post` stores all e

Node 3: The Crea

This node builds a se

```
class CreateIndexNode(Node):
    def prep(self, shared):
        return shared["embeddings"]

    def exec(self, embeddings):
        print("🔍 Creating search index...")
        dimension = embeddings.shape[1]
        index = faiss.IndexFlatL2(dimension)
        index.add(embeddings)
        return index

    def post(self, shared, prep_res, exec_res):
        shared["index"] = exec_res
        print(f"✅ Index created with {exec_res.ntotal} vectors")
        return "default"
```

The `prep` gets embeddings, `exec` creates and populates a vector index, and `post` saves the index for query processing.

Building the Online Flow

Node 4: The EmbedQueryNode

This node converts a user question into the same vector format:

```
class EmbedQueryNode(Node):
    def prep(self, shared):
        return shared["query"]

    def exec(self, query):
        print(f"🔍 Embedding query: {query}")
        query_embedding = get_embedding(query)
        return np

    def post(sel
        shared["
        return "
```

The `prep` gets the user's question, `exec` creates an embedding using the same me as for documents, and `post` stores this embedding for retrieval.

Node 5: The RetrieveDocumentNode

This node finds the most relevant document chunks:

```
class RetrieveDocumentNode(Node):
    def prep(self, shared):
        return shared["query_embedding"], shared["index"],
shared["texts"]

    def exec(self, inputs):
        print("🔎 Searching for relevant documents...")
        query_embedding, index, texts = inputs
        distances, indices = index.search(query_embedding, k=1)
        best_idx = indices[0][0]
        distance = distances[0][0]
        most_relevant_text = texts[best_idx]

        return {
            "text": most_relevant_text,
            "index": best_idx,
            "distance": distance
        }

    def post(self, shared, prep_res, exec_res):
        shared["retrieved_document"] = exec_res
        print(f"📄 Most relevant text: \"{exec_res['text']}\"")
        return "default"
```

The `prep` gets query embedding and index, `exec` searches for the closest matchir document chunk, and `post` saves the retrieved information.

Node 6: The Gene

This node creates th

```
class GenerateAnswerNode(Node):
    def prep(self, shared):
        return shared["query"], shared["retrieved_document"]

    def exec(self, inputs):
        query, retrieved_doc = inputs

        prompt = f"""
Briefly answer the following question based on the context provided:
Question: {query}
Context: {retrieved_doc['text']}
Answer:
"""
        answer = call_llm(prompt)
        return answer

    def post(self, shared, prep_res, exec_res):
        shared["generated_answer"] = exec_res
        print("\n🤖 Generated Answer:")
        print(exec_res)
        return "default"
```

The `prep` gets the question and retrieved document, `exec` creates a prompt and c an LLM, and `post` saves the generated answer.

Connecting Everything Together

```
# Create offline flow for document processing
chunk_docs_node = ChunkDocumentsNode()
embed_docs_node = EmbedDocumentsNode()
create_index_node = CreateIndexNode()

# Connect nodes i
chunk_docs_node >
offline_flow = F

# Create online f
embed_query_node
retrieve_doc_node = RetrieveDocumentNode()
generate_answer_node = GenerateAnswerNode()
```

```
# Connect nodes in sequence
embed_query_node >> retrieve_doc_node >> generate_answer_node
online_flow = Flow(start=embed_query_node)
```

This connects our nodes into two flows - one for document processing (run once) a one for answering questions (run for each query).

The complete working code for this tutorial is available at [GitHub: PocketFlow RAG Cookbook](https://github.com).

Conclusion: The Beauty of RAG Simplicity

Now you understand the elegant simplicity of RAG systems:

1. **Offline Flow**: Process your documents once → Create vectors → Build searcha index
2. **Online Flow**: Process questions → Find relevant context → Generate grounde answers

This simple pattern powers even the most sophisticated RAG systems. Next time y encounter a complex RAG framework, look for these two workflows beneath the abstractions, and you'll understand how it works.

The power of RAG lies in its simplicity and effectiveness. By grounding AI respon in your specific documents, RAG dramatically improves accuracy, reduces hallucinations, and enables AI systems to work with your custom knowledge

With what you've lea ma specific knowledge b

Try Pocket Flow today and experience how 100 lines can replace hundreds of thousan
[GitHub Repository](#) | [Documentation](#) | [TypeScript Version](#)

Thanks for reading Pocket Flow! Subscribe for free to receive new posts and support my work.

20 Likes · 2 Restacks

← Previous Next

Discussion about this post

Comments Restacks

Write a comment...

z4021 Apr 4
Liked by Zachary Huang

I just finished watching the whole thing! It was freakin' awesome! Got me all fired up to make bunch of cool stuff.

LIKE (1) REPLY

1 reply by Zacl

Ian Malcolm Apr
Liked by Zachary

How easy would it be to use an external database for RAG?

LIKE (1) REPLY

1 reply by Zachary Huang

2 more comments...