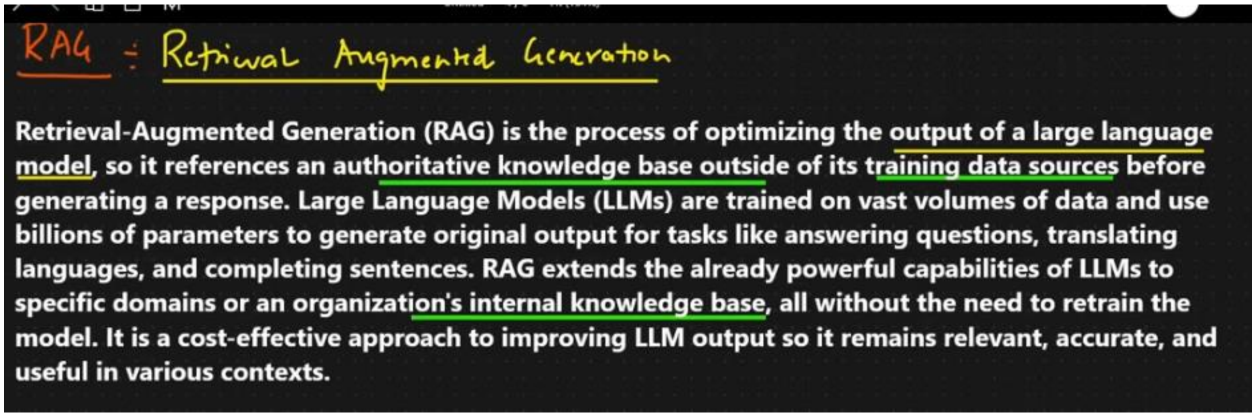
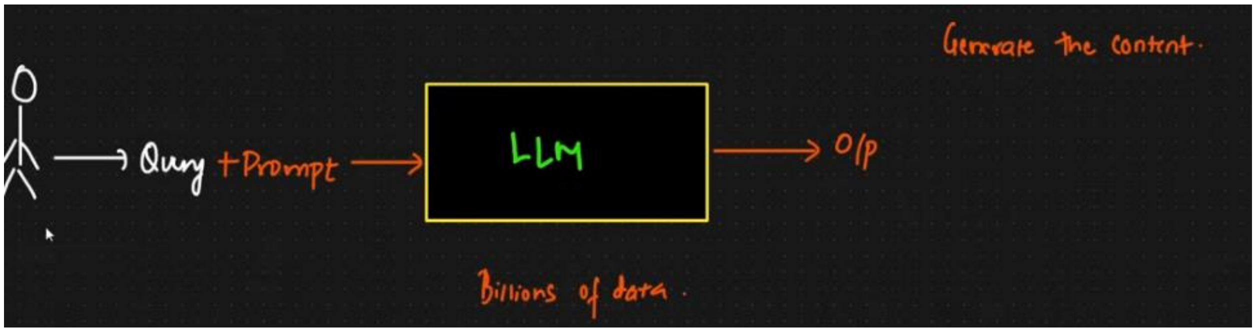
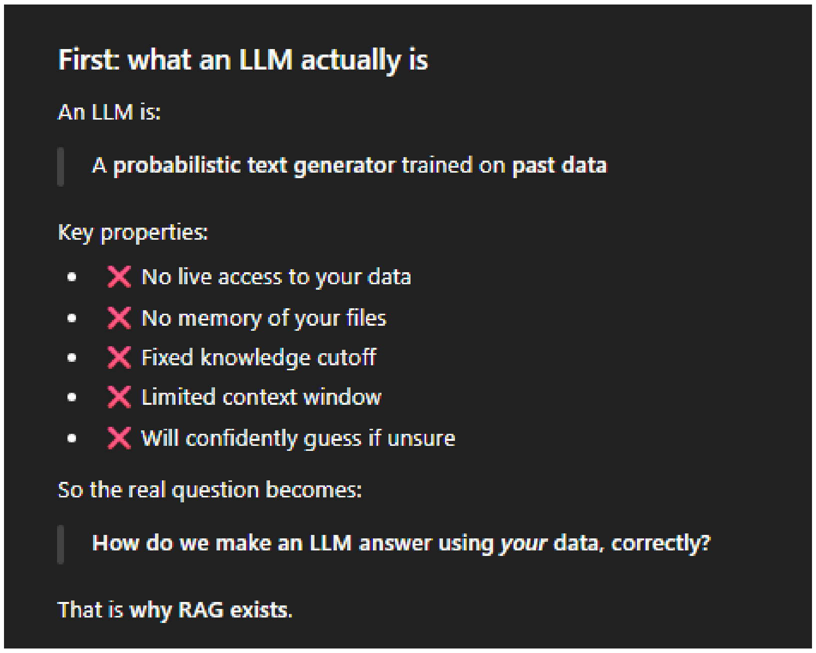
RAG


RAG = Retrival Augmented Generation

Retrieval-Augmented Generation (RAG) is the process of optimizing the output of a large language model, so it references an authoritative knowledge base outside of its training data sources before generating a response. Large Language Models (LLMs) are trained on vast volumes of data and use billions of parameters to generate original output for tasks like answering questions, translating languages, and completing sentences. RAG extends the already powerful capabilities of LLMs to specific domains or an organization's internal knowledge base, all without the need to retrain the model. It is a cost-effective approach to improving LLM output so it remains relevant, accurate, and useful in various contexts.



WHY DO I NEED RAG?


First: what an LLM actually is

An LLM is:

> A **probabilistic text generator** trained on **past data**

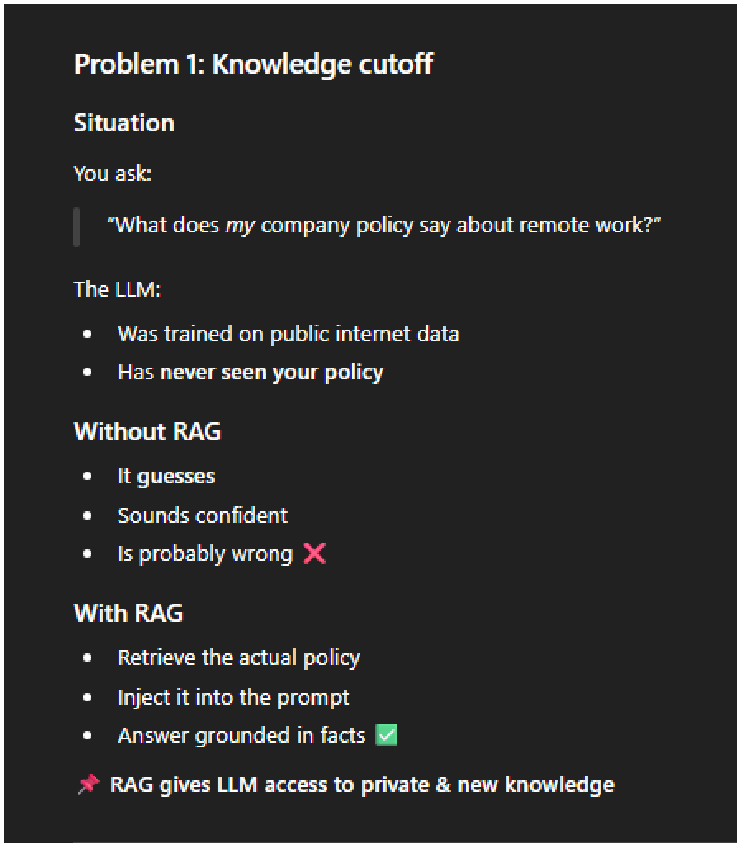
Key properties:

- ❌ No live access to your data
- ❌ No memory of your files
- ❌ Fixed knowledge cutoff
- ❌ Limited context window
- ❌ Will confidently guess if unsure

So the real question becomes:

> **How do we make an LLM answer using *your* data, correctly?**

That is **why RAG exists**.



Problem 1: Knowledge cutoff

Situation

You ask:

> "What does *my* company policy say about remote work?"

The LLM:

- Was trained on public internet data
- Has **never seen your policy**

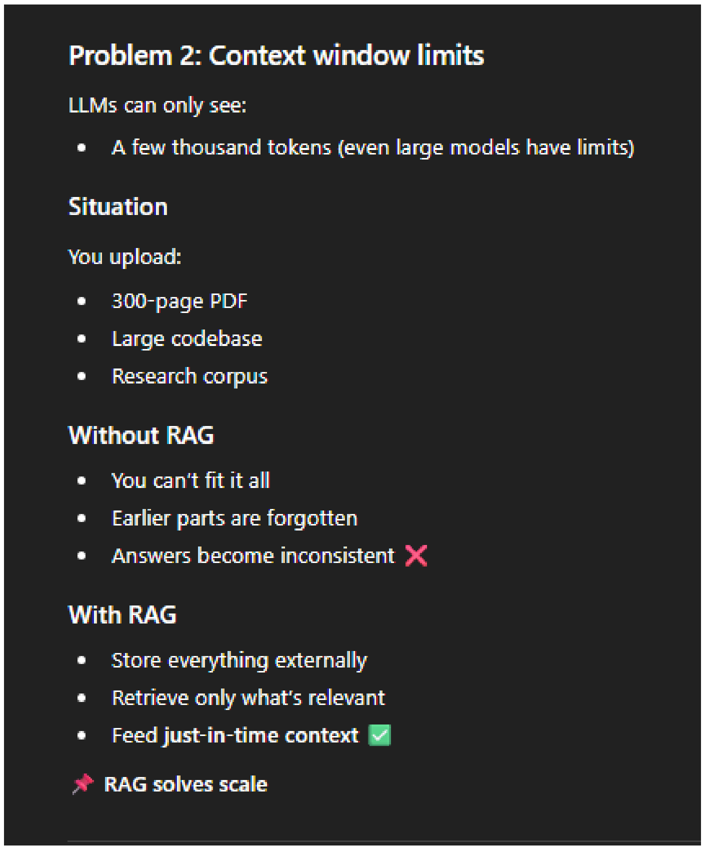
Without RAG

- It **guesses**
- Sounds confident
- Is probably wrong ❌

With RAG

- Retrieve the actual policy
- Inject it into the prompt
- Answer grounded in facts ✅

📌 **RAG gives LLM access to private & new knowledge**



Problem 2: Context window limits

LLMs can only see:

- A few thousand tokens (even large models have limits)

Situation

You upload:

- 300-page PDF
- Large codebase
- Research corpus

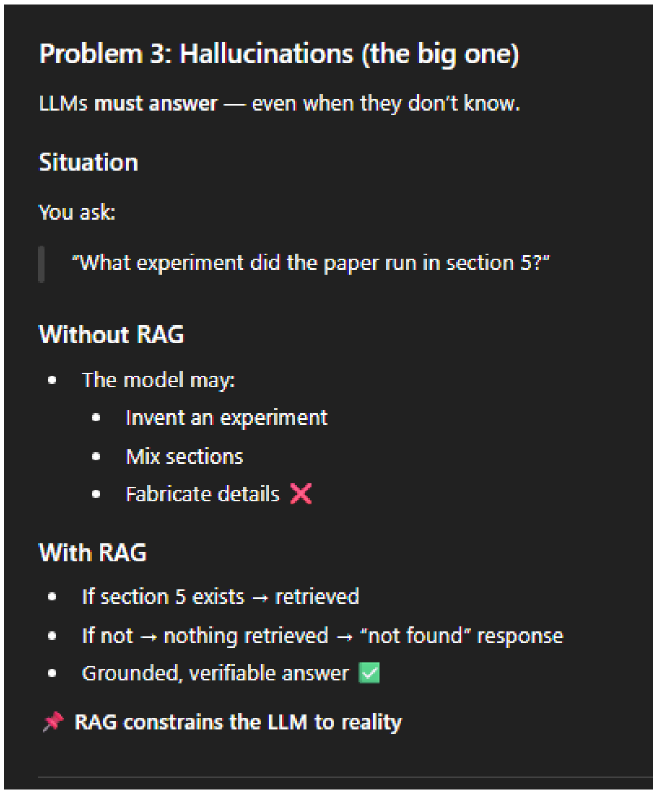
Without RAG

- You can't fit it all
- Earlier parts are forgotten
- Answers become inconsistent ❌

With RAG

- Store everything externally
- Retrieve only what's relevant
- Feed **just-in-time context** ✅

📌 **RAG solves scale**



Problem 3: Hallucinations (the big one)

LLMs **must answer** — even when they don't know.

Situation

You ask:

> "What experiment did the paper run in section 5?"

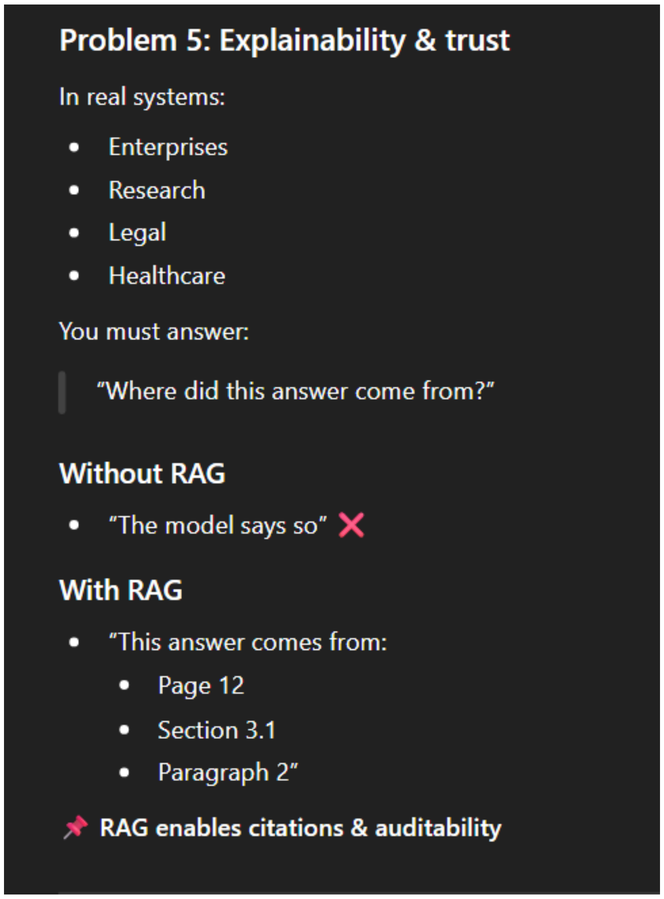
Without RAG

- The model may:
  - Invent an experiment
  - Mix sections
  - Fabricate details ❌

With RAG

- If section 5 exists → retrieved
- If not → nothing retrieved → "not found" response
- Grounded, verifiable answer ✅

📌 **RAG constrains the LLM to reality**



Problem 5: Explainability & trust

In real systems:

- Enterprises
- Research
- Legal
- Healthcare

You must answer:

> "Where did this answer come from?"

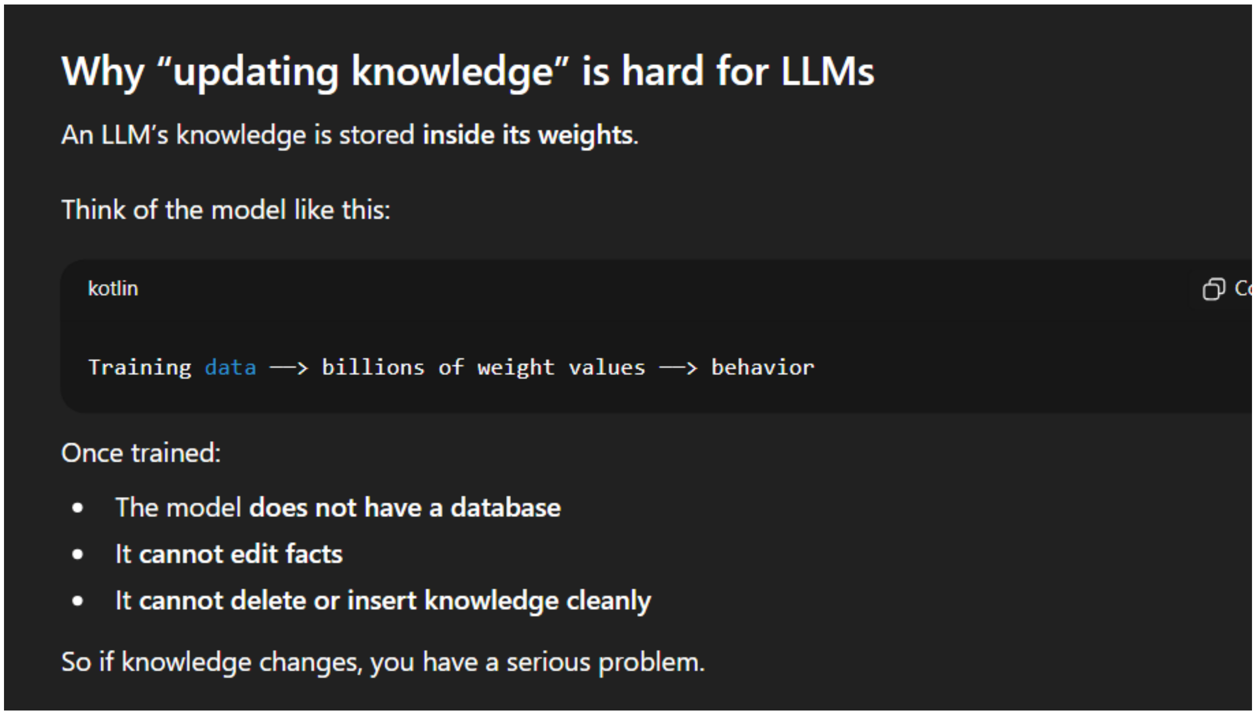
Without RAG

- "The model says so" ❌

With RAG

- "This answer comes from:
  - Page 12
  - Section 3.1
  - Paragraph 2"

📌 **RAG enables citations & auditability**



Why "updating knowledge" is hard for LLMs

An LLM's knowledge is stored **inside its weights**.

Think of the model like this:

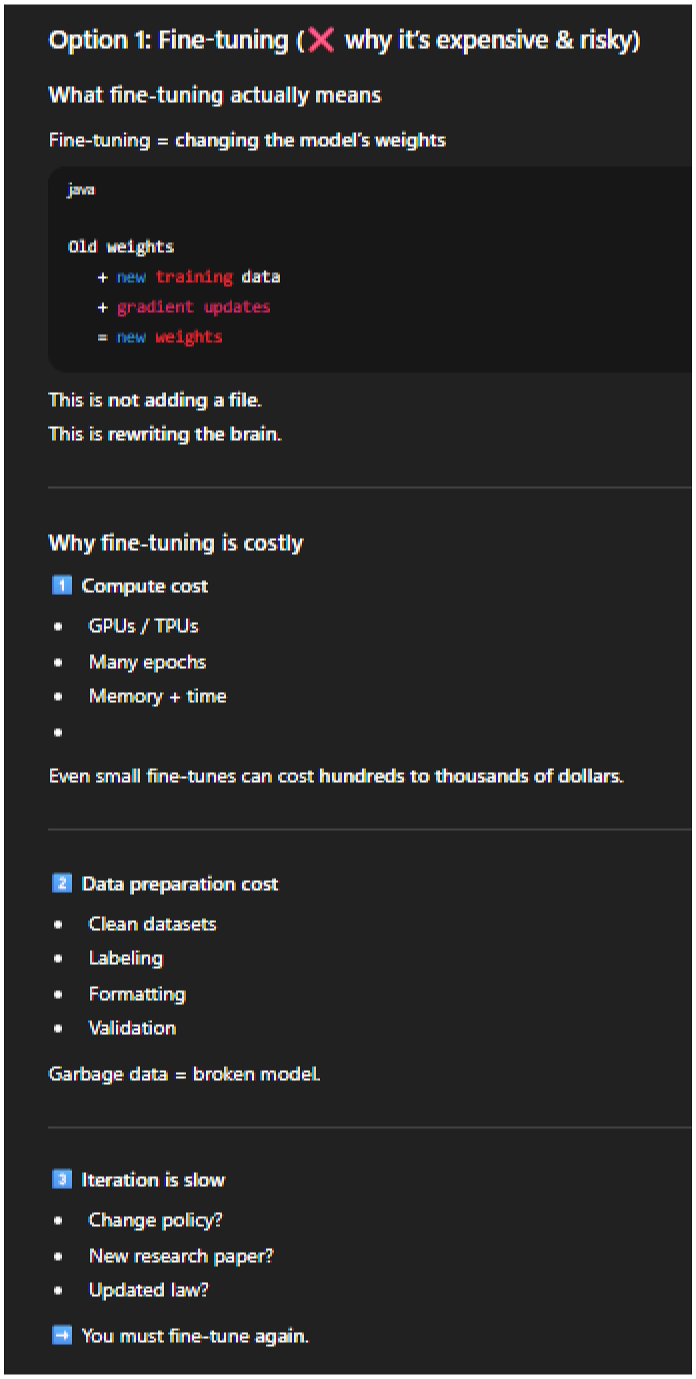
kotlin

```kotlin
Training data ──> billions of weight values ──> behavior
```

Once trained:

- The model **does not have a database**
- It **cannot edit facts**
- It **cannot delete or insert knowledge cleanly**

So if knowledge changes, you have a serious problem.



Option 1: Fine-tuning (❌ why it's expensive & risky)

What fine-tuning actually means

Fine-tuning = changing the model's weights

```java
Old weights
    + new training data
    + gradient updates
    = new weights
```

This is not adding a file.
This is rewriting the brain.

Why fine-tuning is costly

1️⃣ Compute cost

- GPUs / TPUs
- Many epochs
- Memory + time
-

Even small fine-tunes can cost hundreds to thousands of dollars.

2️⃣ Data preparation cost

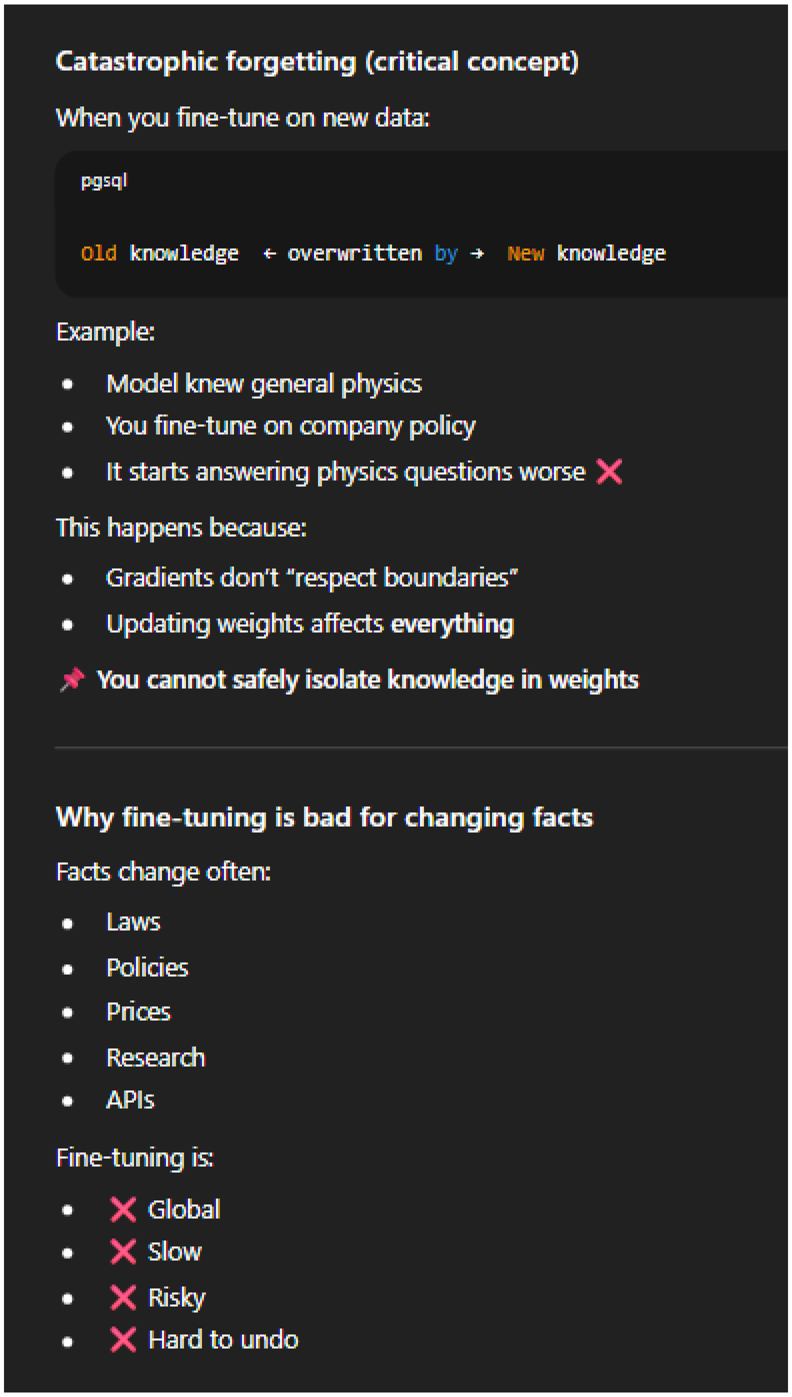
- Clean datasets
- Labeling
- Formatting
- Validation

Garbage data = broken model.

3️⃣ Iteration is slow

- Change policy?
- New research paper?
- Updated law?

➡️ You must fine-tune again.



Catastrophic forgetting (critical concept)

When you fine-tune on new data:

```pgsql
Old knowledge ← overwritten by → New knowledge
```

Example:

- Model knew general physics
- You fine-tune on company policy
- It starts answering physics questions worse ❌

This happens because:

- Gradients don't "respect boundaries"
- Updating weights affects **everything**

📌 **You cannot safely isolate knowledge in weights**

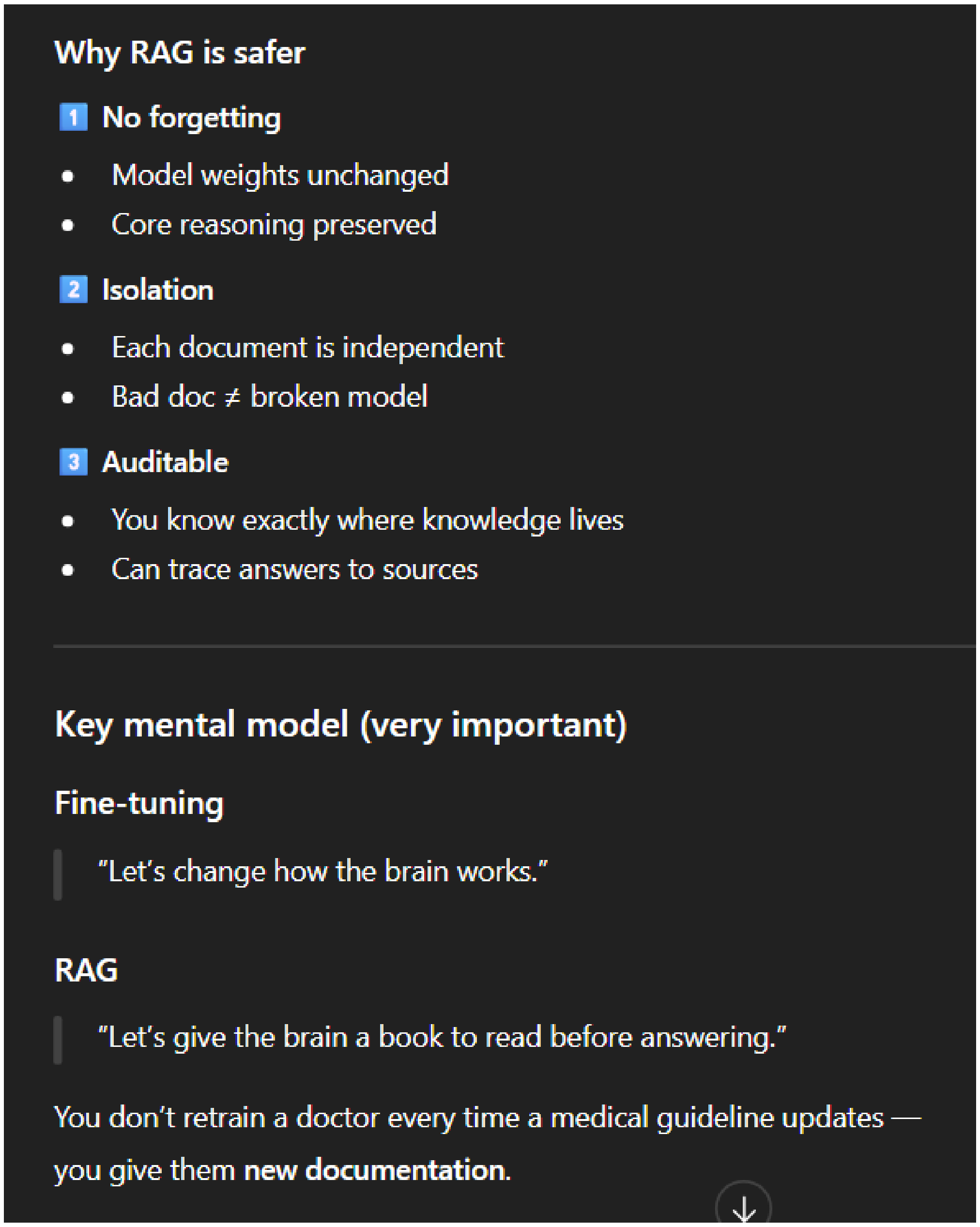
Why fine-tuning is bad for changing facts

Facts change often:

- Laws
- Policies
- Prices
- Research
- APIs

Fine-tuning is:

- ❌ Global
- ❌ Slow
- ❌ Risky
- ❌ Hard to undo



Why RAG is safer

1️⃣ No forgetting

- Model weights unchanged
- Core reasoning preserved

2️⃣ Isolation

- Each document is independent
- Bad doc ≠ broken model

3️⃣ Auditable

- You know exactly where knowledge lives
- Can trace answers to sources

Key mental model (very important)

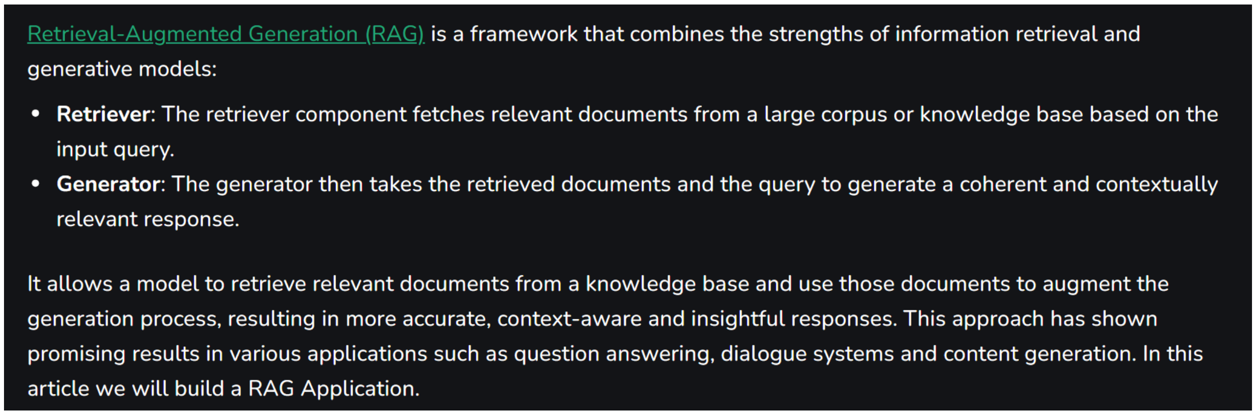
Fine-tuning

> "Let's change how the brain works."

RAG

> "Let's give the brain a book to read before answering."

You don't retrain a doctor every time a medical guideline updates — you give them **new documentation**.

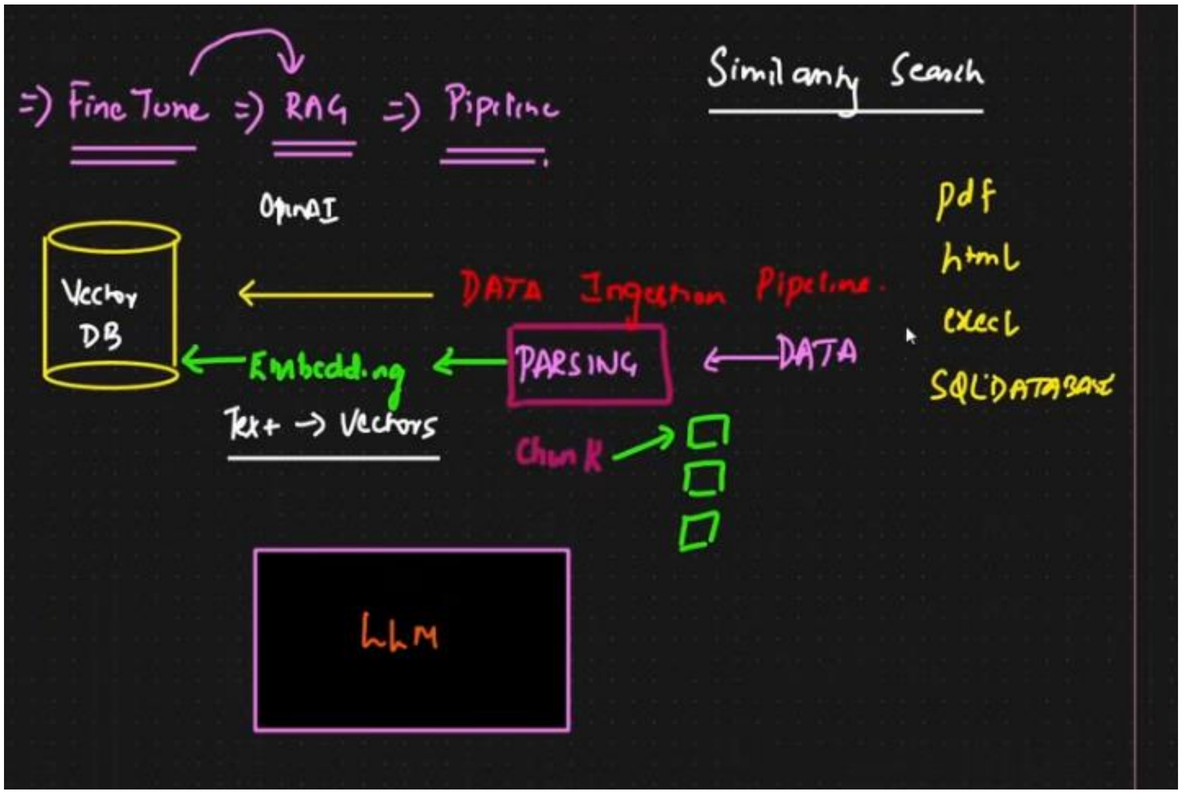


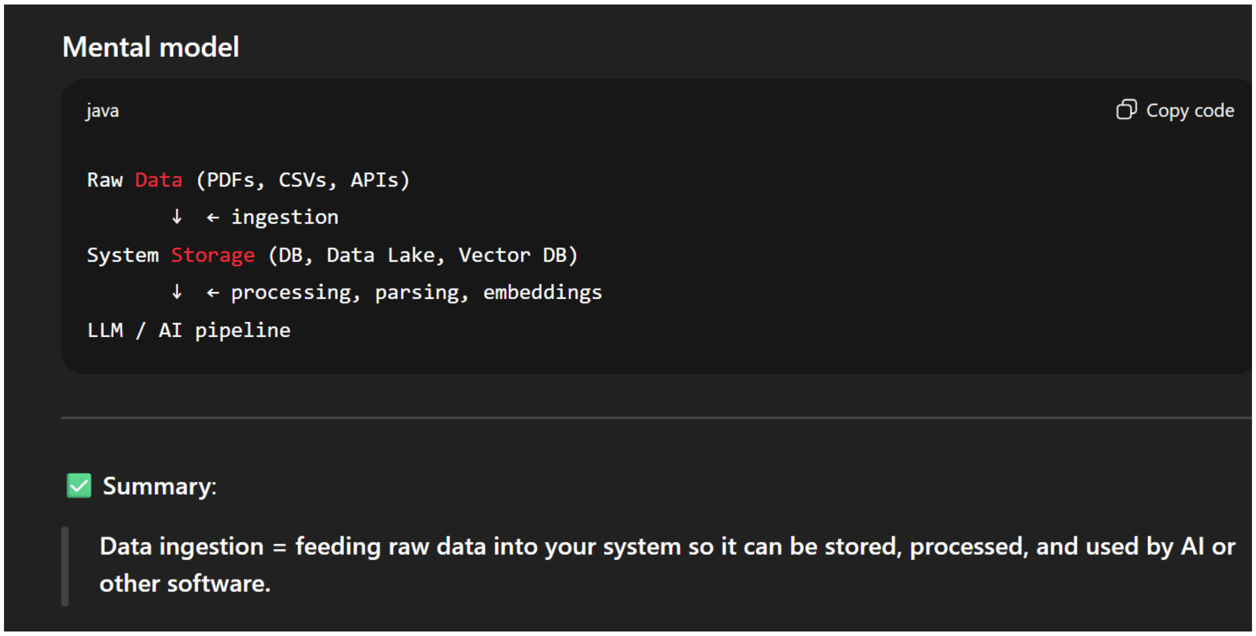
Retrieval-Augmented Generation (RAG) is a framework that combines the strengths of information retrieval and generative models:

- **Retriever**: The retriever component fetches relevant documents from a large corpus or knowledge base based on the input query.
- **Generator**: The generator then takes the retrieved documents and the query to generate a coherent and contextually relevant response.

It allows a model to retrieve relevant documents from a knowledge base and use those documents to augment the generation process, resulting in more accurate, context-aware and insightful responses. This approach has shown promising results in various applications such as question answering, dialogue systems and content generation. In this article we will build a RAG Application.

DATA INGETION





Mental model

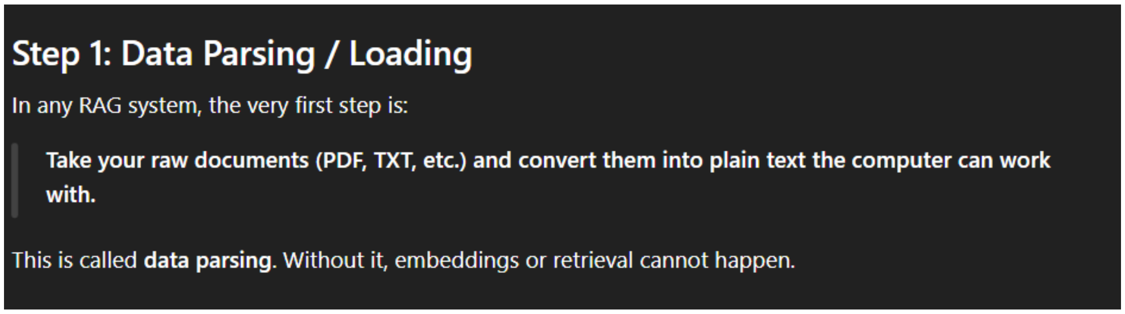
```java
Raw Data (PDFs, CSVs, APIs)
       ↓  ← ingestion
System Storage (DB, Data Lake, Vector DB)
       ↓  ← processing, parsing, embeddings
LLM / AI pipeline
```

✅ **Summary**:

> Data ingestion = feeding raw data into your system so it can be stored, processed, and used by AI or other software.

STEP -1

DATA PARSING:

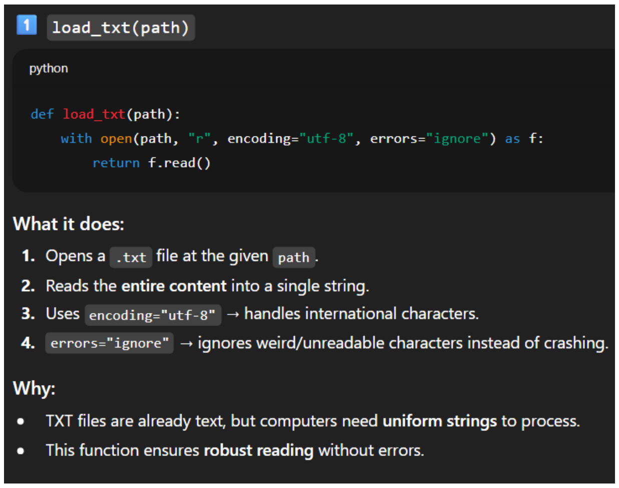


Step 1: Data Parsing / Loading

In any RAG system, the very first step is:

> **Take your raw documents (PDF, TXT, etc.) and convert them into plain text the computer can work with.**

This is called **data parsing**. Without it, embeddings or retrieval cannot happen.



1️⃣ `load_txt(path)`

python

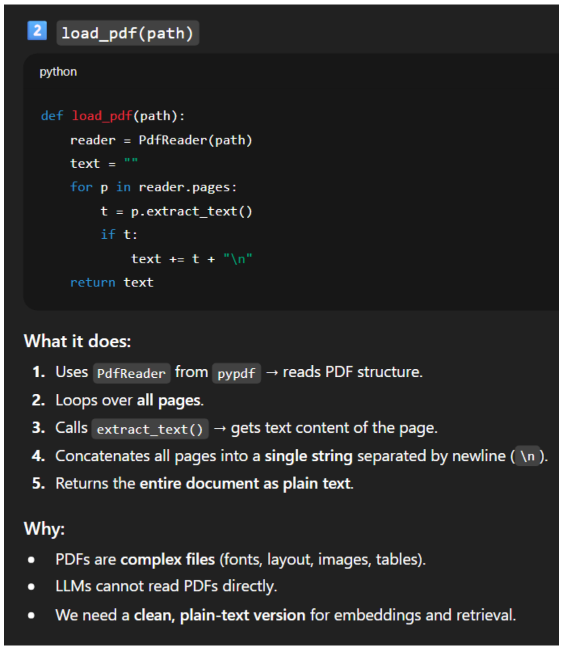
```python
def load_txt(path):
    with open(path, "r", encoding="utf-8", errors="ignore") as f:
        return f.read()
```

What it does:

1. Opens a `.txt` file at the given `path`.
2. Reads the **entire content** into a single string.
3. Uses `encoding="utf-8"` → handles international characters.
4. `errors="ignore"` → ignores weird/unreadable characters instead of crashing.

Why:

- TXT files are already text, but computers need **uniform strings** to process.
- This function ensures **robust reading** without errors.



2️⃣ `load_pdf(path)`

python

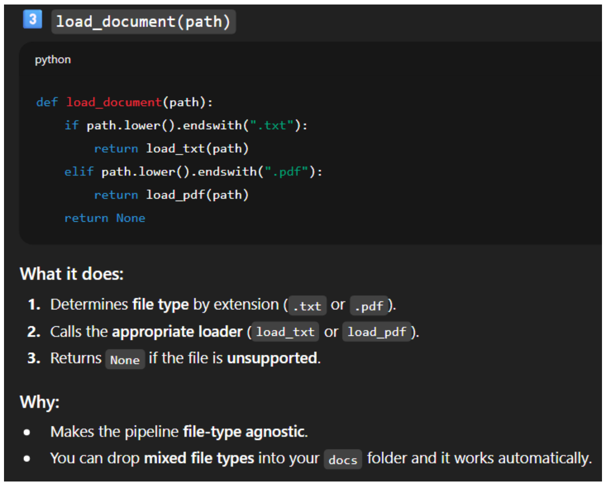
```python
def load_pdf(path):
    reader = PdfReader(path)
    text = ""
    for p in reader.pages:
        t = p.extract_text()
        if t:
            text += t + "\n"
    return text
```

What it does:

1. Uses `PdfReader` from `pypdf` → reads PDF structure.
2. Loops over **all pages**.
3. Calls `extract_text()` → gets text content of the page.
4. Concatenates all pages into a **single string** separated by newline (`\n`).
5. Returns the **entire document as plain text**.

Why:

- PDFs are **complex files** (fonts, layout, images, tables).
- LLMs cannot read PDFs directly.
- We need a **clean, plain-text version** for embeddings and retrieval.



3 `load_document(path)`

```python
def load_document(path):
    if path.lower().endswith(".txt"):
        return load_txt(path)
    elif path.lower().endswith(".pdf"):
        return load_pdf(path)
    return None
```

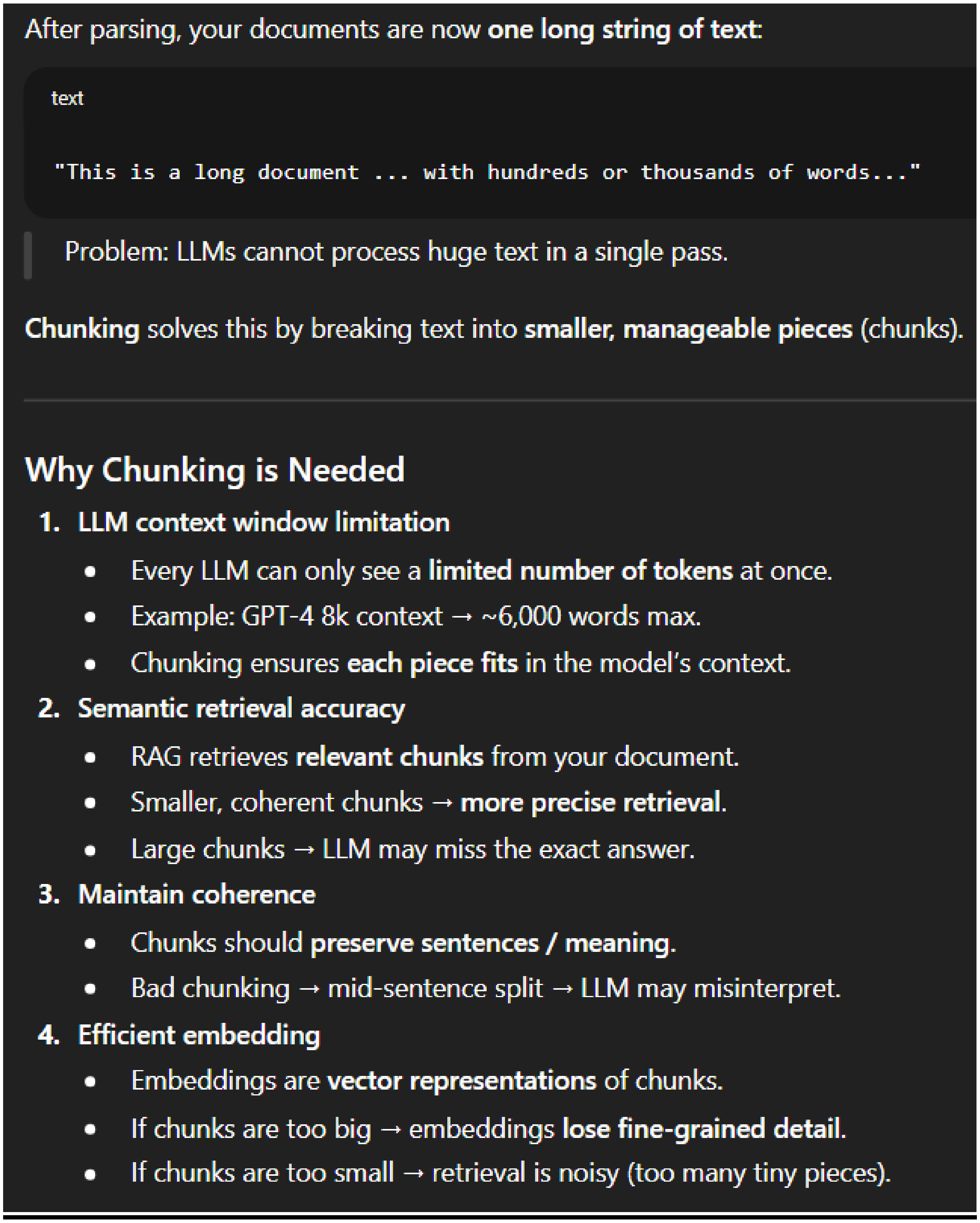
What it does:

1. Determines **file type** by extension (`.txt` or `.pdf`).
2. Calls the **appropriate loader** (`load_txt` or `load_pdf`).
3. Returns `None` if the file is **unsupported**.

Why:

- Makes the pipeline **file-type agnostic**.
- You can drop **mixed file types** into your `docs` folder and it works automatically.

Step 2: Chunking Process



After parsing, your documents are now **one long string of text**:

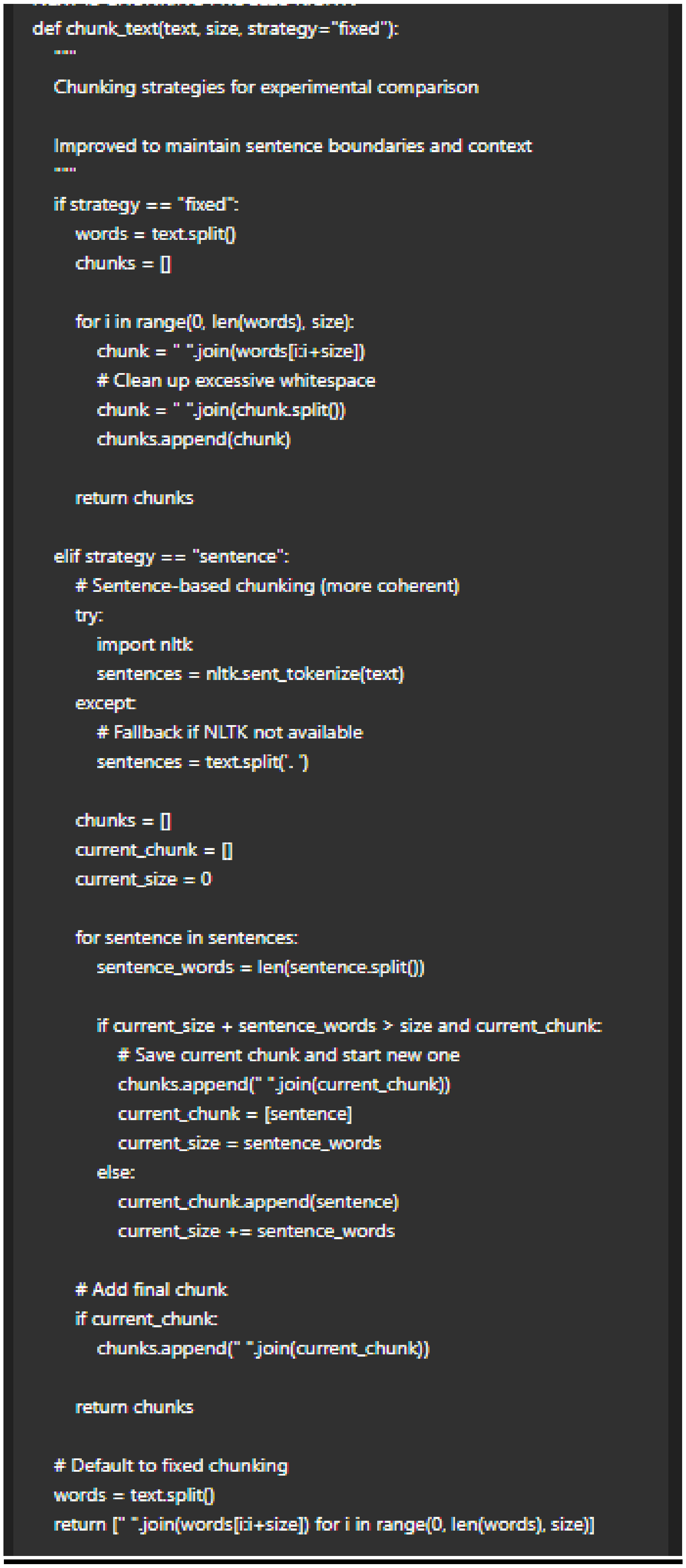
```text
"This is a long document ... with hundreds or thousands of words..."
```

> Problem: LLMs cannot process huge text in a single pass.

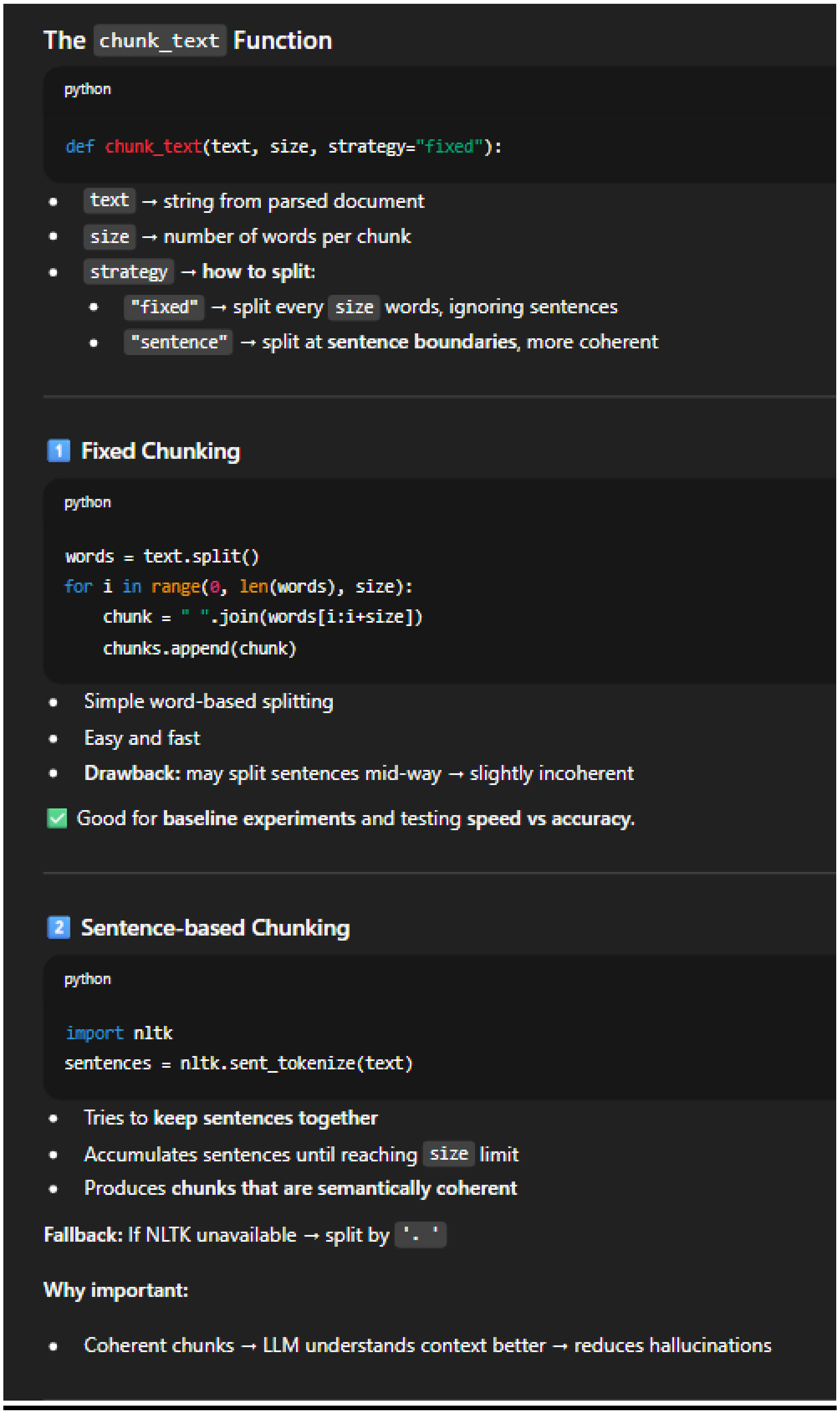
Chunking solves this by breaking text into **smaller, manageable pieces** (chunks).

Why Chunking is Needed

1. **LLM context window limitation**
   - Every LLM can only see a **limited number of tokens** at once.
   - Example: GPT-4 8k context → ~6,000 words max.
   - Chunking ensures **each piece fits** in the model's context.
2. **Semantic retrieval accuracy**
   - RAG retrieves **relevant chunks** from your document.
   - Smaller, coherent chunks → **more precise retrieval**.
   - Large chunks → LLM may miss the exact answer.
3. **Maintain coherence**
   - Chunks should **preserve sentences / meaning**.
   - Bad chunking → mid-sentence split → LLM may misinterpret.
4. **Efficient embedding**
   - Embeddings are **vector representations** of chunks.
   - If chunks are too big → embeddings **lose fine-grained detail**.
   - If chunks are too small → retrieval is noisy (too many tiny pieces).



```
def chunk_text(text, size, strategy="fixed"):
    """
    Chunking strategies for experimental comparison

    Improved to maintain sentence boundaries and context
    """
    if strategy == "fixed":
        words = text.split()
        chunks = []

        for i in range(0, len(words), size):
            chunk = " ".join(words[i:i+size])
            # Clean up excessive whitespace
            chunk = " ".join(chunk.split())
            chunks.append(chunk)

        return chunks

    elif strategy == "sentence":
        # Sentence-based chunking (more coherent)
        try:
            import nltk
            sentences = nltk.sent_tokenize(text)
        except:
            # Fallback if NLTK not available
            sentences = text.split('. ')

        chunks = []
        current_chunk = []
        current_size = 0

        for sentence in sentences:
            sentence_words = len(sentence.split())

            if current_size + sentence_words > size and current_chunk:
                # Save current chunk and start new one
                chunks.append(" ".join(current_chunk))
                current_chunk = [sentence]
                current_size = sentence_words
            else:
                current_chunk.append(sentence)
                current_size += sentence_words

        # Add final chunk
        if current_chunk:
            chunks.append(" ".join(current_chunk))

        return chunks

    # Default to fixed chunking
    words = text.split()
    return [" ".join(words[i:i+size]) for i in range(0, len(words), size)]
```



The `chunk_text` Function

```python
def chunk_text(text, size, strategy="fixed"):
```

- `text` → string from parsed document
- `size` → number of words per chunk
- `strategy` → **how to split**:
  - `"fixed"` → split every `size` words, ignoring sentences
  - `"sentence"` → split at **sentence boundaries**, more coherent

1️⃣ Fixed Chunking

```python
words = text.split()
for i in range(0, len(words), size):
    chunk = " ".join(words[i:i+size])
    chunks.append(chunk)
```

- Simple word-based splitting
- Easy and fast
- **Drawback:** may split sentences mid-way → slightly incoherent

✅ Good for **baseline experiments** and testing **speed vs accuracy**.

2️⃣ Sentence-based Chunking

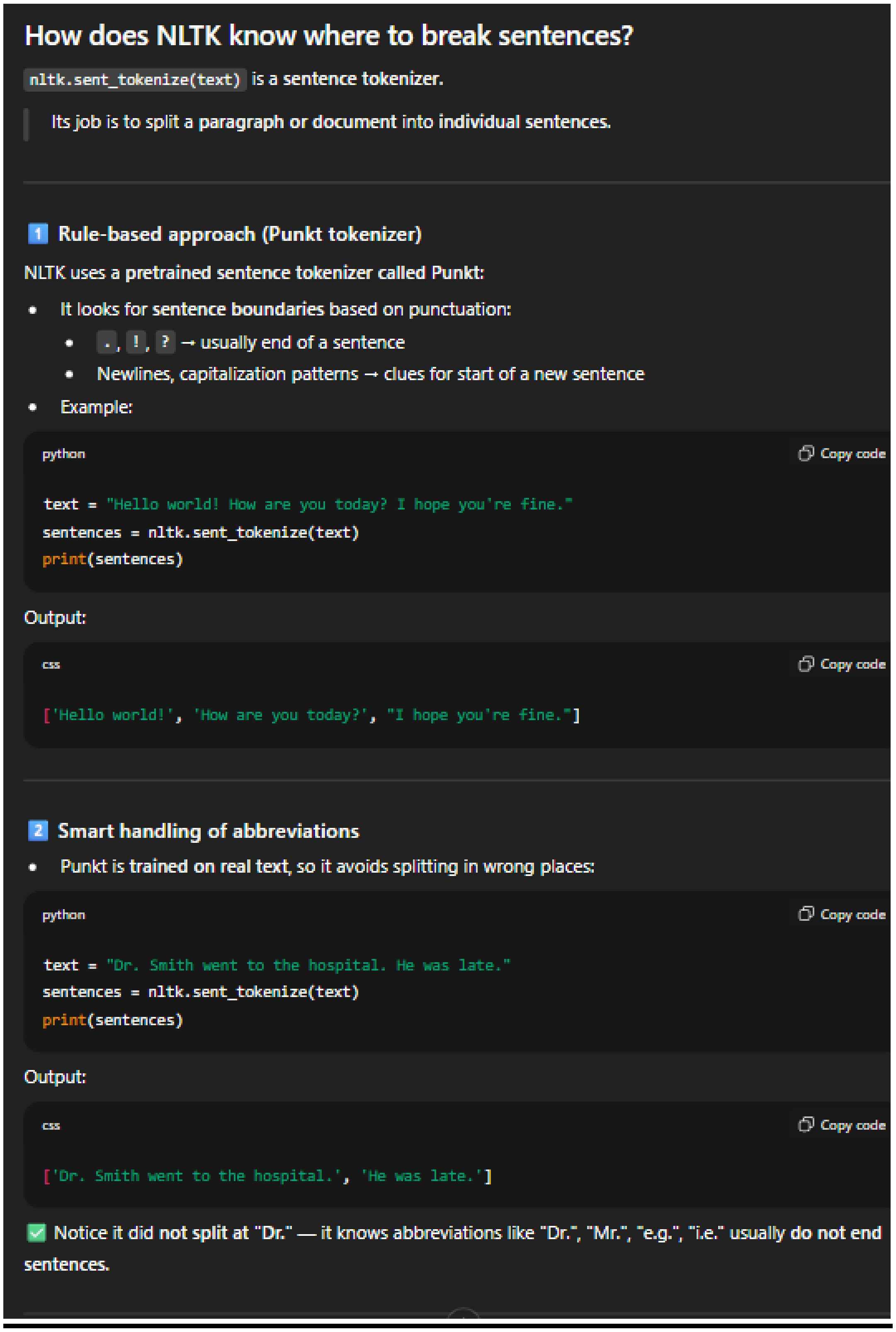
```python
import nltk
sentences = nltk.sent_tokenize(text)
```

- Tries to **keep sentences together**
- Accumulates sentences until reaching `size` limit
- Produces **chunks that are semantically coherent**

Fallback: If NLTK unavailable → split by `'. '`

Why important:

- Coherent chunks → LLM understands context better → reduces hallucinations



How does NLTK know where to break sentences?

`nltk.sent_tokenize(text)` is a **sentence tokenizer**.

> Its job is to split a **paragraph or document** into **individual sentences**.

1️⃣ Rule-based approach (Punkt tokenizer)

NLTK uses a **pretrained sentence tokenizer called Punkt**:

- It looks for **sentence boundaries** based on punctuation:
  - `.`, `!`, `?` → usually end of a sentence
  - Newlines, capitalization patterns → clues for start of a new sentence
- Example:

```python
text = "Hello world! How are you today? I hope you're fine."
sentences = nltk.sent_tokenize(text)
print(sentences)
```

Output:

```css
['Hello world!', 'How are you today?', "I hope you're fine."]
```

2️⃣ Smart handling of abbreviations

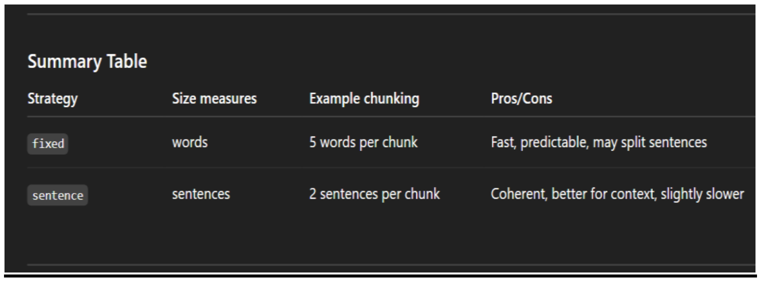
- Punkt is **trained on real text**, so it avoids splitting in wrong places:

```python
text = "Dr. Smith went to the hospital. He was late."
sentences = nltk.sent_tokenize(text)
print(sentences)
```

Output:

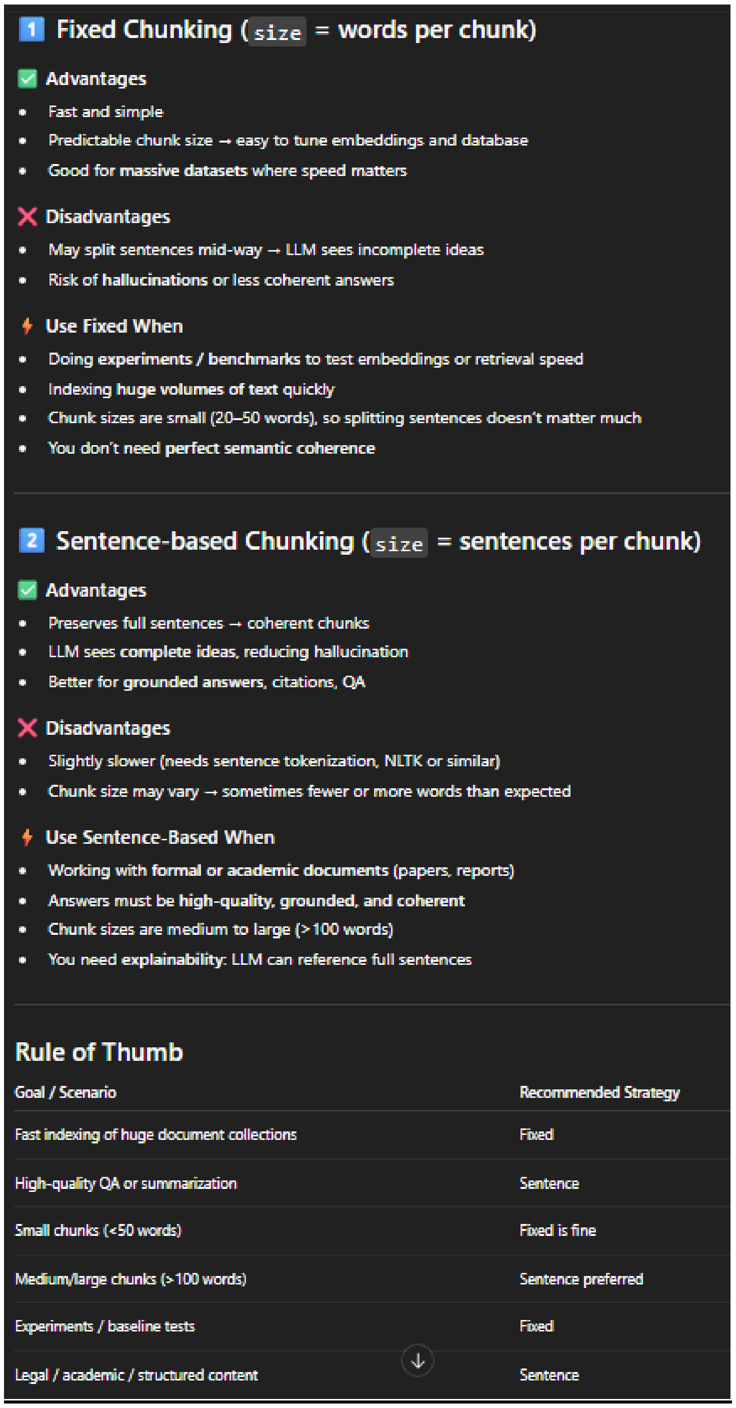
```css
['Dr. Smith went to the hospital.', 'He was late.']
```

✅ Notice it **did not split at "Dr."** — it knows abbreviations like "Dr.", "Mr.", "e.g.", "i.e." usually **do not end sentences**.



Summary Table

Strategy	Size measures	Example chunking	Pros/Cons
`fixed`	words	5 words per chunk	Fast, predictable, may split sentences
`sentence`	sentences	2 sentences per chunk	Coherent, better for context, slightly slower



1️⃣ Fixed Chunking (`size` = words per chunk)

✅ Advantages

- Fast and simple
- Predictable chunk size → easy to tune embeddings and database
- Good for **massive datasets** where speed matters

❌ Disadvantages

- May split sentences mid-way → LLM sees incomplete ideas
- Risk of **hallucinations** or less coherent answers

⚡ Use Fixed When

- Doing **experiments / benchmarks** to test embeddings or retrieval speed
- Indexing **huge volumes of text** quickly
- Chunk sizes are small (20–50 words), so splitting sentences doesn't matter much
- You don't need **perfect semantic coherence**

2️⃣ Sentence-based Chunking (`size` = sentences per chunk)

✅ Advantages

- Preserves full sentences → coherent chunks
- LLM sees **complete ideas**, reducing hallucination
- Better for **grounded answers**, citations, QA

❌ Disadvantages

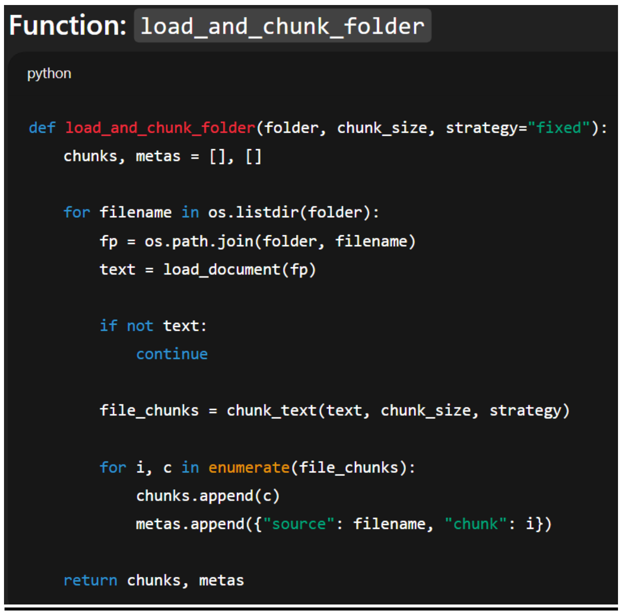
- Slightly slower (needs sentence tokenization, NLTK or similar)
- Chunk size may vary → sometimes fewer or more words than expected

⚡ Use Sentence-Based When

- Working with **formal or academic documents** (papers, reports)
- Answers must be **high-quality, grounded, and coherent**
- Chunk sizes are medium to large (>100 words)
- You need **explainability**: LLM can reference full sentences

Rule of Thumb

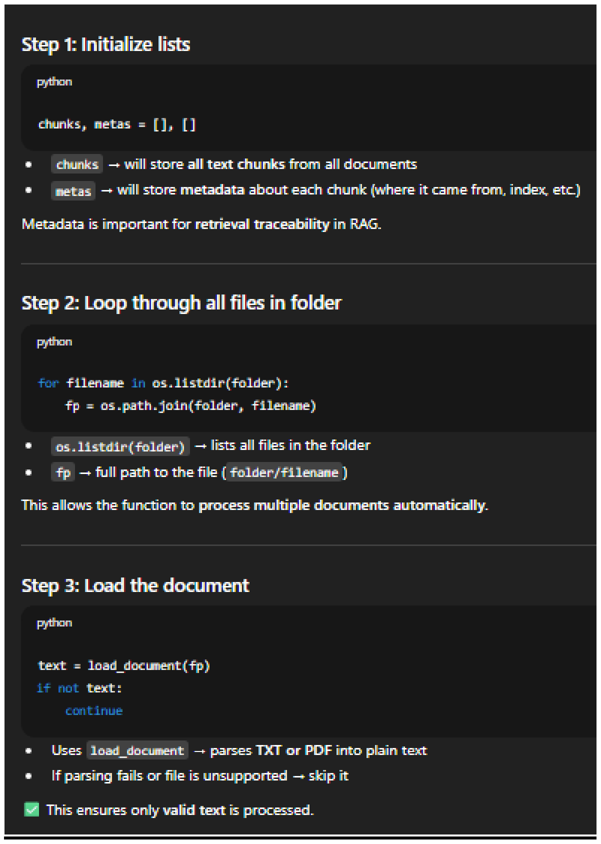
Goal / Scenario	Recommended Strategy
Fast indexing of huge document collections	Fixed
High-quality QA or summarization	Sentence
Small chunks (<50 words)	Fixed is fine
Medium/large chunks (>100 words)	Sentence preferred
Experiments / baseline tests	Fixed
Legal / academic / structured content	Sentence



Function: `load_and_chunk_folder`

python

```python
def load_and_chunk_folder(folder, chunk_size, strategy="fixed"):
    chunks, metas = [], []

    for filename in os.listdir(folder):
        fp = os.path.join(folder, filename)
        text = load_document(fp)

        if not text:
            continue

        file_chunks = chunk_text(text, chunk_size, strategy)

        for i, c in enumerate(file_chunks):
            chunks.append(c)
            metas.append({"source": filename, "chunk": i})

    return chunks, metas
```



Step 1: Initialize lists

python

```python
chunks, metas = [], []
```

- `chunks` → will store **all text chunks** from all documents
- `metas` → will store **metadata** about each chunk (where it came from, index, etc.)

Metadata is important for **retrieval traceability** in RAG.

Step 2: Loop through all files in folder

python

```python
for filename in os.listdir(folder):
    fp = os.path.join(folder, filename)
```

- `os.listdir(folder)` → lists all files in the folder
- `fp` → full path to the file (`folder/filename`)

This allows the function to **process multiple documents automatically**.

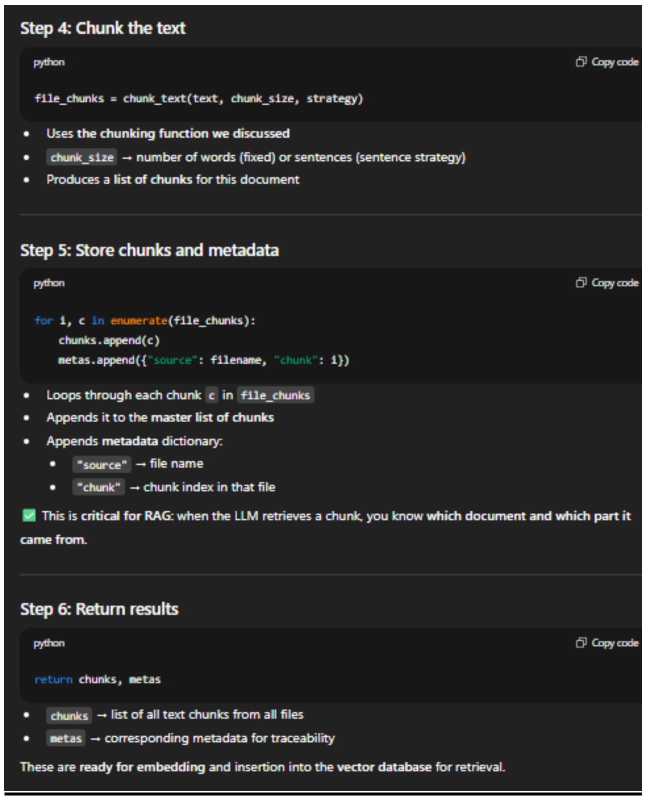
Step 3: Load the document

python

```python
text = load_document(fp)
if not text:
    continue
```

- Uses `load_document` → parses **TXT or PDF** into plain text
- If parsing fails or file is unsupported → skip it

✅ This ensures only **valid text** is processed.



Step 4: Chunk the text

python

```python
file_chunks = chunk_text(text, chunk_size, strategy)
```

- Uses the **chunking function** we discussed
- `chunk_size` → number of words (fixed) or sentences (sentence strategy)
- Produces a **list of chunks** for this document

Step 5: Store chunks and metadata

python

```python
for i, c in enumerate(file_chunks):
    chunks.append(c)
    metas.append({"source": filename, "chunk": i})
```

- Loops through each chunk `c` in `file_chunks`
- Appends it to the **master list of chunks**
- Appends **metadata** dictionary:
  - `"source"` → file name
  - `"chunk"` → chunk index in that file

✅ This is **critical for RAG**: when the LLM retrieves a chunk, you know **which document and which part** it came from.

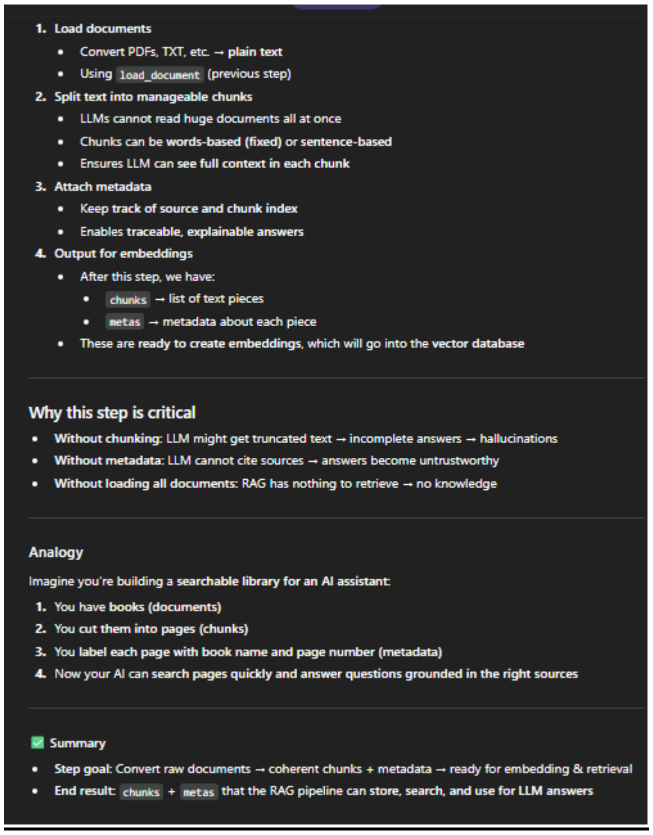
Step 6: Return results

python

```python
return chunks, metas
```

- `chunks` → list of all text chunks from all files
- `metas` → corresponding metadata for traceability

These are **ready for embedding** and insertion into the **vector database** for retrieval.



1. **Load documents**
   - Convert PDFs, TXT, etc. → **plain text**
   - Using `load_document` (previous step)
2. **Split text into manageable chunks**
   - LLMs cannot read huge documents all at once
   - Chunks can be **words-based (fixed)** or **sentence-based**
   - Ensures LLM can see **full context in each chunk**
3. **Attach metadata**
   - Keep track of **source and chunk index**
   - Enables **traceable, explainable answers**
4. **Output for embeddings**
   - After this step, we have:
     - `chunks` → list of text pieces
     - `metas` → metadata about each piece
   - These are **ready to create embeddings**, which will go into the **vector database**

Why this step is critical

- **Without chunking:** LLM might get truncated text → incomplete answers → hallucinations
- **Without metadata:** LLM cannot cite sources → answers become untrustworthy
- **Without loading all documents:** RAG has nothing to retrieve → no knowledge

Analogy

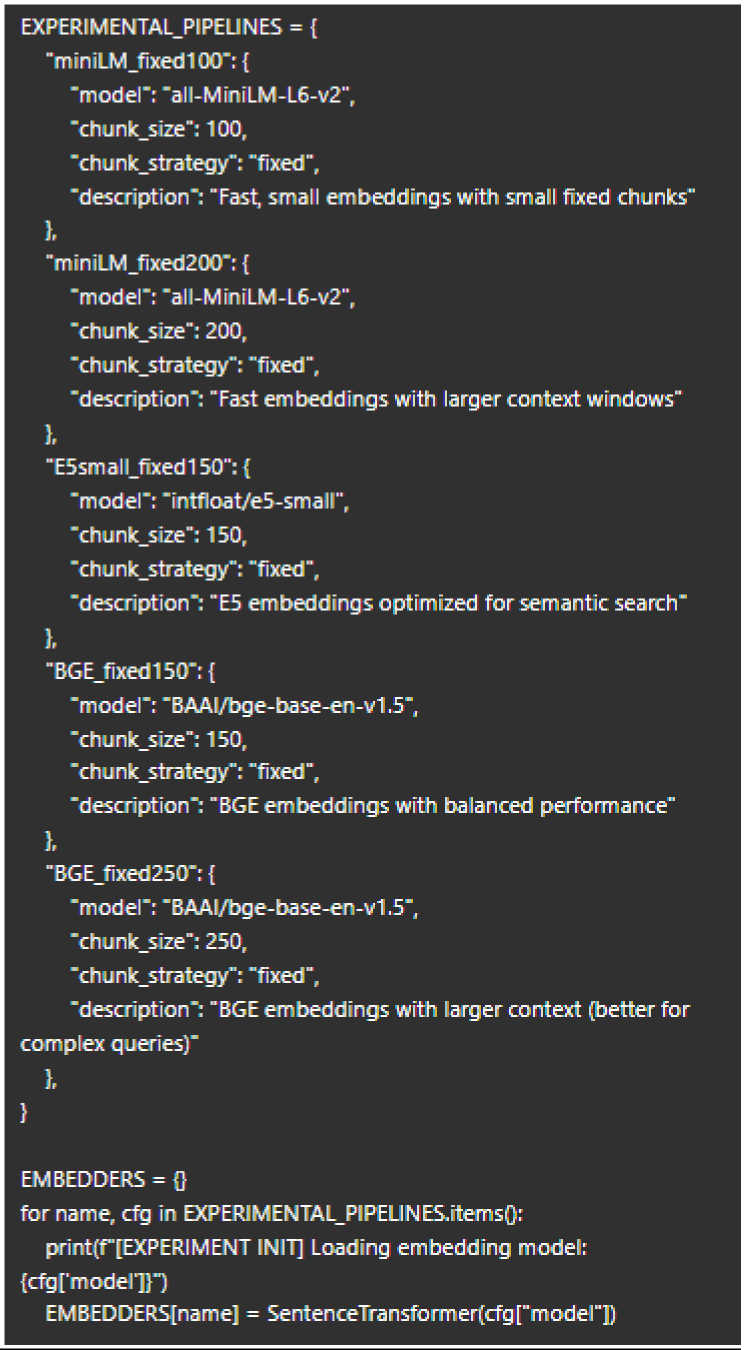
Imagine you're building a **searchable library for an AI assistant**:

1. You have **books (documents)**
2. You **cut them into pages (chunks)**
3. You **label each page with book name and page number (metadata)**
4. Now your AI can **search pages quickly and answer questions grounded in the right sources**

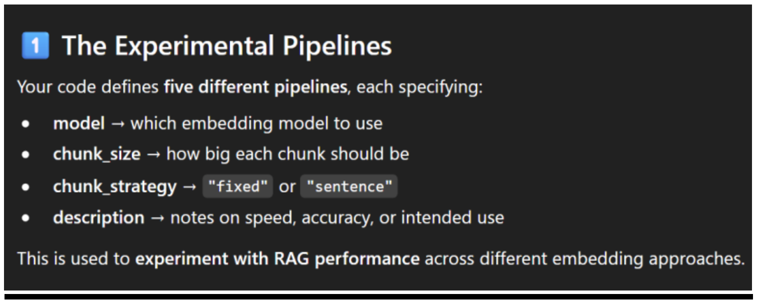
✅ **Summary**

- **Step goal:** Convert raw documents → coherent chunks + metadata → ready for embedding & retrieval
- **End result:** `chunks` + `metas` that the RAG pipeline can **store, search, and use for LLM answers**

Step 3: Embedding



```
EXPERIMENTAL_PIPELINES = {
    "miniLM_fixed100": {
        "model": "all-MiniLM-L6-v2",
        "chunk_size": 100,
        "chunk_strategy": "fixed",
        "description": "Fast, small embeddings with small fixed chunks"
    },
    "miniLM_fixed200": {
        "model": "all-MiniLM-L6-v2",
        "chunk_size": 200,
        "chunk_strategy": "fixed",
        "description": "Fast embeddings with larger context windows"
    },
    "E5small_fixed150": {
        "model": "intfloat/e5-small",
        "chunk_size": 150,
        "chunk_strategy": "fixed",
        "description": "E5 embeddings optimized for semantic search"
    },
    "BGE_fixed150": {
        "model": "BAAI/bge-base-en-v1.5",
        "chunk_size": 150,
        "chunk_strategy": "fixed",
        "description": "BGE embeddings with balanced performance"
    },
    "BGE_fixed250": {
        "model": "BAAI/bge-base-en-v1.5",
        "chunk_size": 250,
        "chunk_strategy": "fixed",
        "description": "BGE embeddings with larger context (better for
complex queries)"
    },
}

EMBEDDERS = {}
for name, cfg in EXPERIMENTAL_PIPELINES.items():
    print(f"[EXPERIMENT INIT] Loading embedding model:
{cfg['model']}")
    EMBEDDERS[name] = SentenceTransformer(cfg["model"])
```

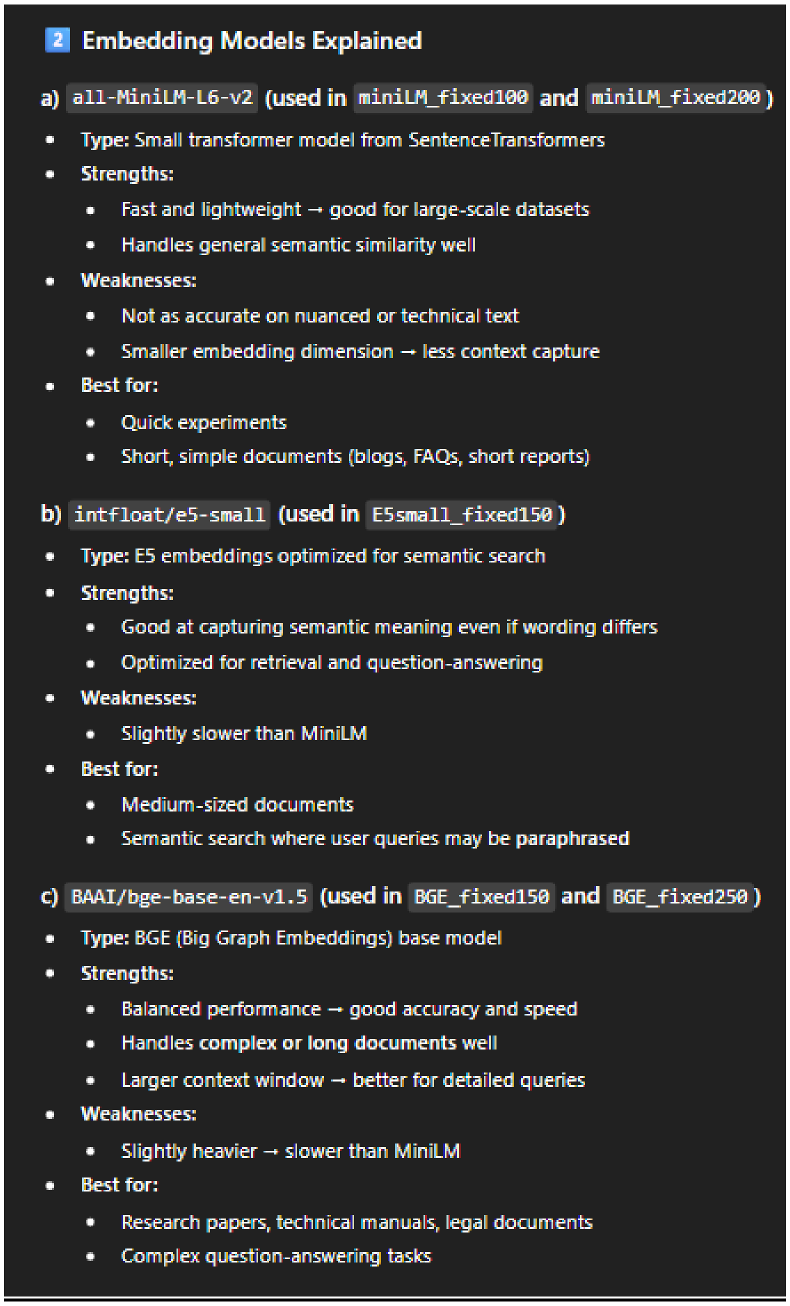


1️⃣ The Experimental Pipelines

Your code defines **five different pipelines**, each specifying:

- **model** → which embedding model to use
- **chunk_size** → how big each chunk should be
- **chunk_strategy** → `"fixed"` or `"sentence"`
- **description** → notes on speed, accuracy, or intended use

This is used to **experiment with RAG performance** across different embedding approaches.



2️⃣ Embedding Models Explained

a) `all-MiniLM-L6-v2` (used in `miniLM_fixed100` and `miniLM_fixed200`)

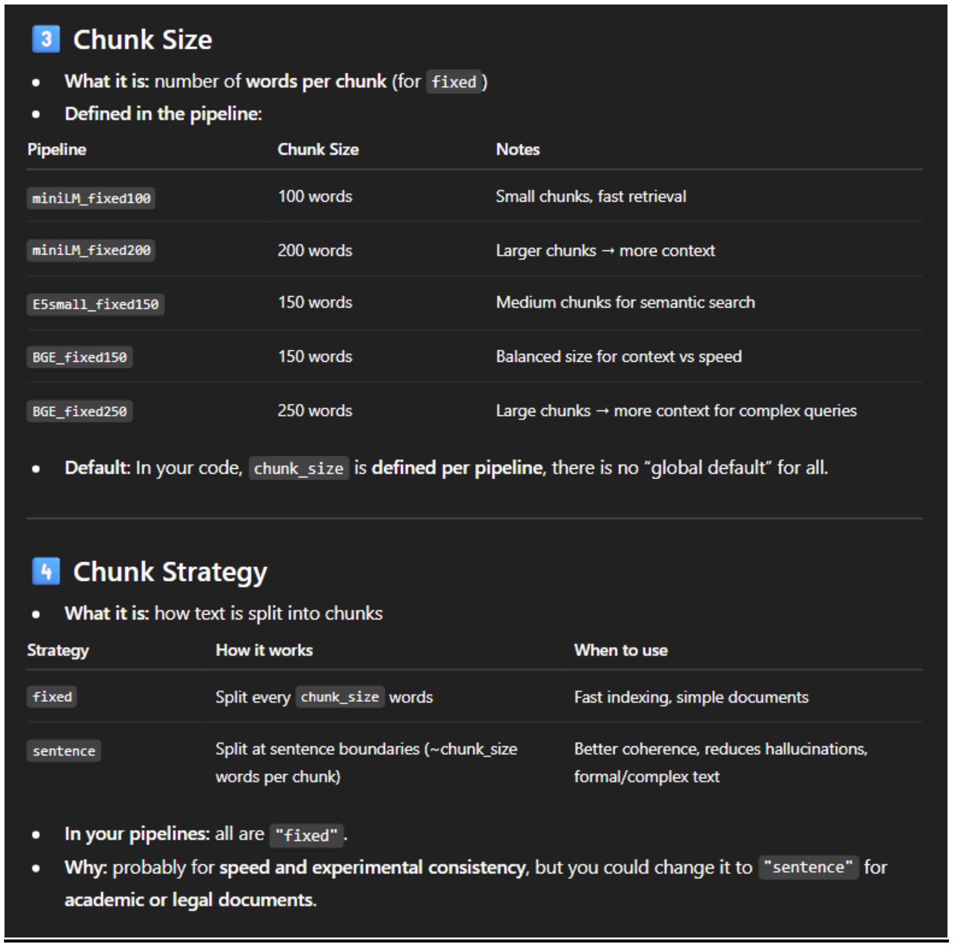
- **Type:** Small transformer model from SentenceTransformers
- **Strengths:**
  - Fast and lightweight → good for large-scale datasets
  - Handles general semantic similarity well
- **Weaknesses:**
  - Not as accurate on nuanced or technical text
  - Smaller embedding dimension → less context capture
- **Best for:**
  - Quick experiments
  - Short, simple documents (blogs, FAQs, short reports)

b) `intfloat/e5-small` (used in `E5small_fixed150`)

- **Type:** E5 embeddings optimized for semantic search
- **Strengths:**
  - Good at capturing semantic meaning even if wording differs
  - Optimized for retrieval and question-answering
- **Weaknesses:**
  - Slightly slower than MiniLM
- **Best for:**
  - Medium-sized documents
  - Semantic search where user queries may be **paraphrased**

c) `BAAI/bge-base-en-v1.5` (used in `BGE_fixed150` and `BGE_fixed250`)

- **Type:** BGE (Big Graph Embeddings) base model
- **Strengths:**
  - Balanced performance → good accuracy and speed
  - Handles **complex or long documents** well
  - Larger context window → better for detailed queries
- **Weaknesses:**
  - Slightly heavier → slower than MiniLM
- **Best for:**
  - Research papers, technical manuals, legal documents
  - Complex question-answering tasks



3️⃣ Chunk Size

- **What it is:** number of **words per chunk** (for `fixed`)
- **Defined in the pipeline:**

Pipeline	Chunk Size	Notes
`miniLM_fixed100`	100 words	Small chunks, fast retrieval
`miniLM_fixed200`	200 words	Larger chunks → more context
`E5small_fixed150`	150 words	Medium chunks for semantic search
`BGE_fixed150`	150 words	Balanced size for context vs speed
`BGE_fixed250`	250 words	Large chunks → more context for complex queries

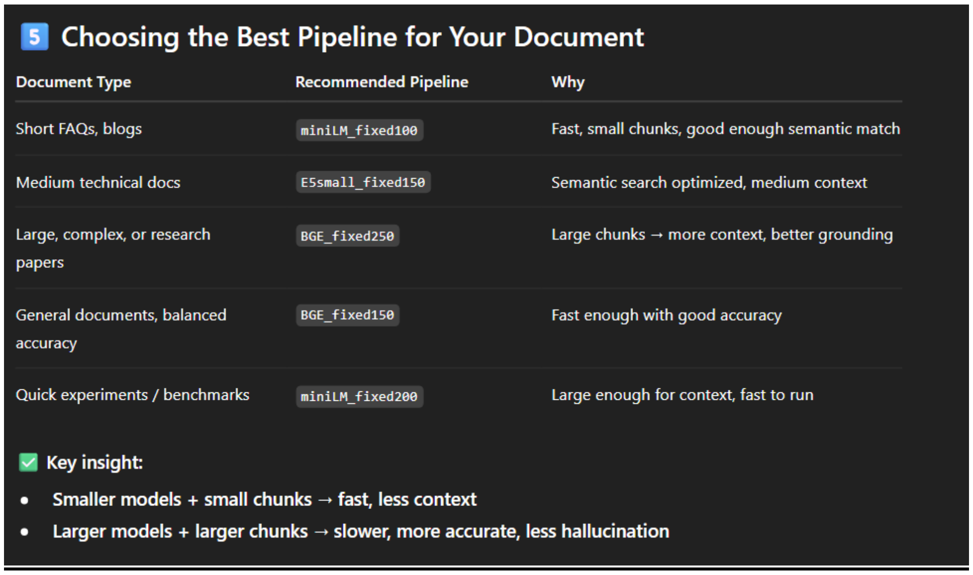
- **Default:** In your code, `chunk_size` is **defined per pipeline**, there is no "global default" for all.

4️⃣ Chunk Strategy

- **What it is:** how text is split into chunks

Strategy	How it works	When to use
`fixed`	Split every `chunk_size` words	Fast indexing, simple documents
`sentence`	Split at sentence boundaries (~chunk_size words per chunk)	Better coherence, reduces hallucinations, formal/complex text

- **In your pipelines:** all are `"fixed"`.
- **Why:** probably for **speed and experimental consistency**, but you could change it to `"sentence"` for **academic or legal documents.**

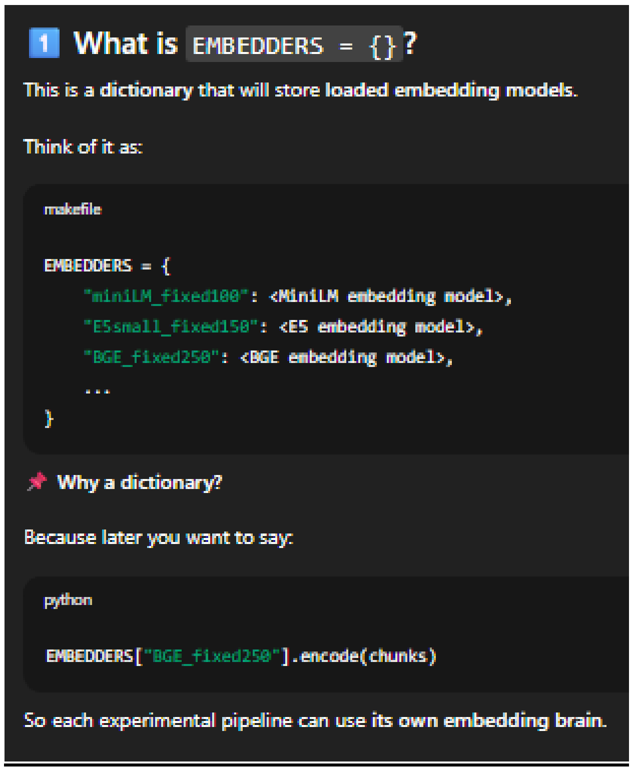


5️⃣ Choosing the Best Pipeline for Your Document

Document Type	Recommended Pipeline	Why
Short FAQs, blogs	`miniLM_fixed100`	Fast, small chunks, good enough semantic match
Medium technical docs	`E5small_fixed150`	Semantic search optimized, medium context
Large, complex, or research papers	`BGE_fixed250`	Large chunks → more context, better grounding
General documents, balanced accuracy	`BGE_fixed150`	Fast enough with good accuracy
Quick experiments / benchmarks	`miniLM_fixed200`	Large enough for context, fast to run

✅ **Key insight:**

- **Smaller models + small chunks → fast, less context**
- **Larger models + larger chunks → slower, more accurate, less hallucination**



1 What is `EMBEDDERS = {}`?

This is a dictionary that will store loaded embedding models.

Think of it as:

makefile

```
EMBEDDERS = {
    "miniLM_fixed100": <MiniLM embedding model>,
    "E5small_fixed150": <E5 embedding model>,
    "BGE_fixed250": <BGE embedding model>,
    ...
}
```

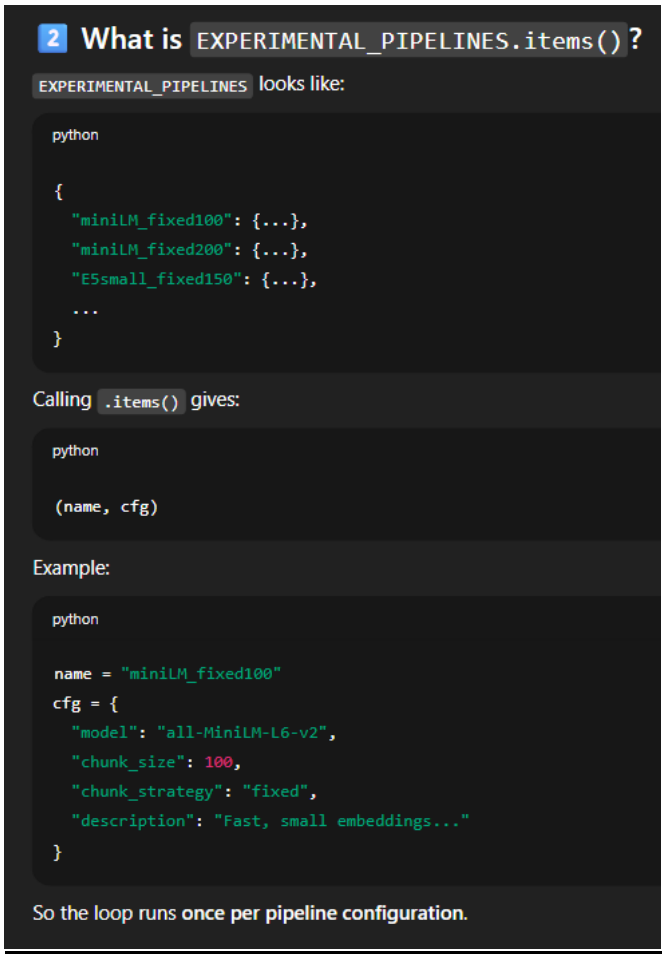
📌 **Why a dictionary?**

Because later you want to say:

python

```
EMBEDDERS["BGE_fixed250"].encode(chunks)
```

So each experimental pipeline can use its own embedding brain.



2 What is `EXPERIMENTAL_PIPELINES.items()`?

`EXPERIMENTAL_PIPELINES` looks like:

python

```
{
  "miniLM_fixed100": {...},
  "miniLM_fixed200": {...},
  "E5small_fixed150": {...},
  ...
}
```

Calling `.items()` gives:

python

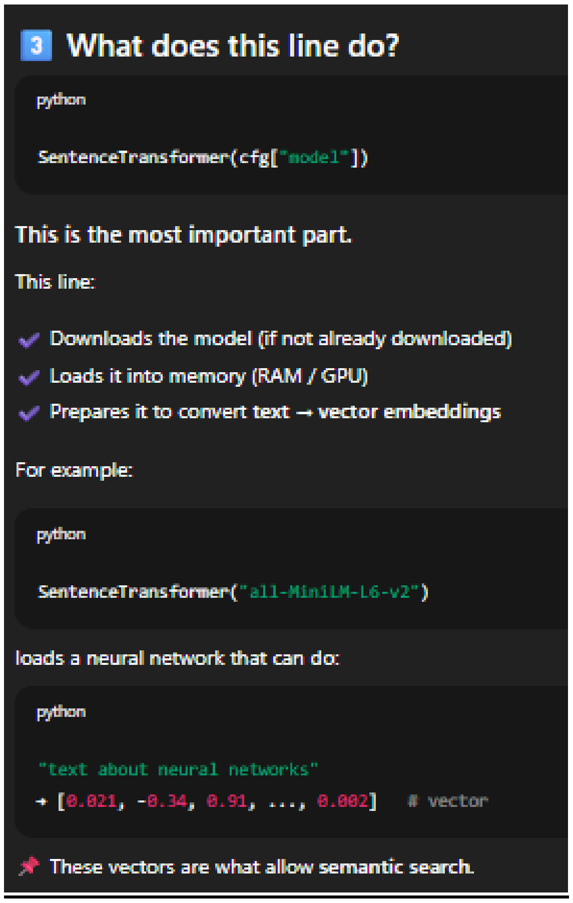
```
(name, cfg)
```

Example:

python

```
name = "miniLM_fixed100"
cfg = {
  "model": "all-MiniLM-L6-v2",
  "chunk_size": 100,
  "chunk_strategy": "fixed",
  "description": "Fast, small embeddings..."
}
```

So the loop runs **once per pipeline configuration.**



3️⃣ What does this line do?

```python
SentenceTransformer(cfg["model"])
```

This is the most important part.

This line:

✔ Downloads the model (if not already downloaded)
✔ Loads it into memory (RAM / GPU)
✔ Prepares it to convert text → vector embeddings

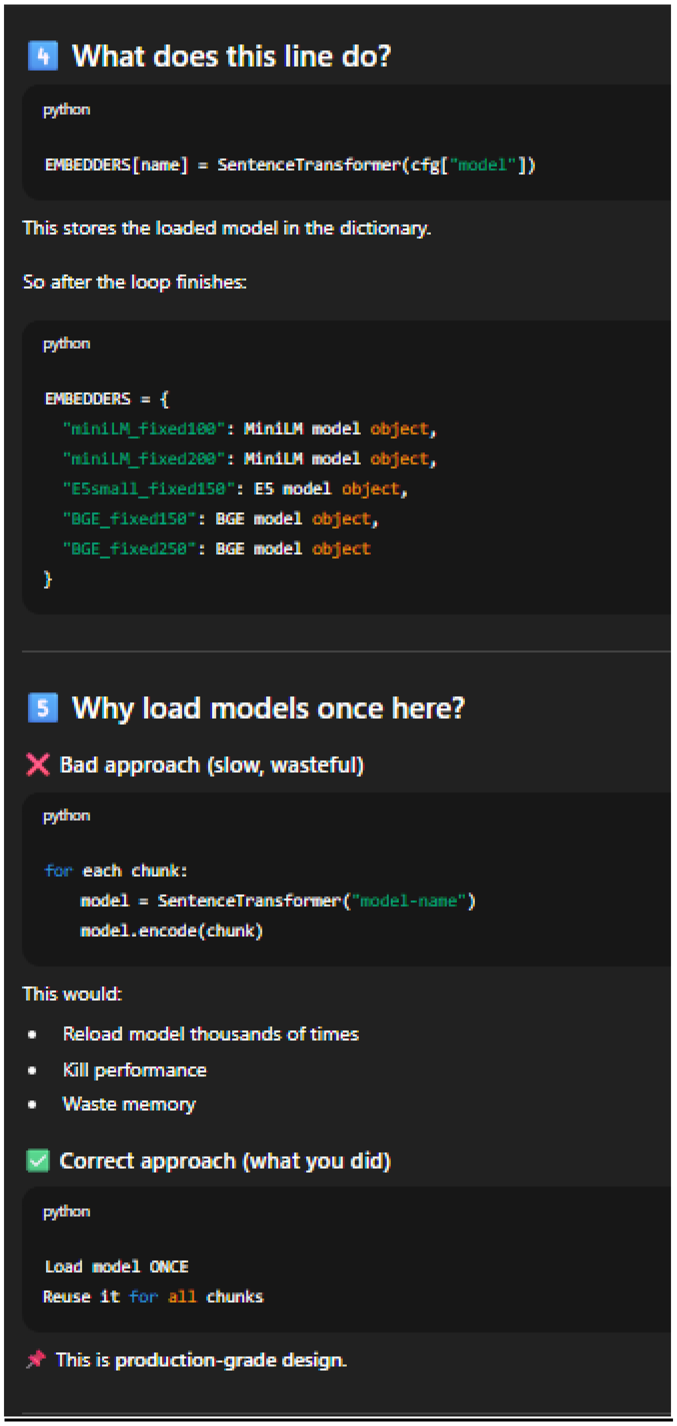
For example:

```python
SentenceTransformer("all-MiniLM-L6-v2")
```

loads a neural network that can do:

```python
"text about neural networks"
→ [0.021, -0.34, 0.91, ..., 0.002]   # vector
```

📌 These vectors are what allow semantic search.



4️⃣ What does this line do?

```python
EMBEDDERS[name] = SentenceTransformer(cfg["model"])
```

This stores the loaded model in the dictionary.

So after the loop finishes:

```python
EMBEDDERS = {
  "miniLM_fixed100": MiniLM model object,
  "miniLM_fixed200": MiniLM model object,
  "E5small_fixed150": E5 model object,
  "BGE_fixed150": BGE model object,
  "BGE_fixed250": BGE model object
}
```

5️⃣ Why load models once here?

❌ **Bad approach (slow, wasteful)**

```python
for each chunk:
    model = SentenceTransformer("model-name")
    model.encode(chunk)
```

This would:

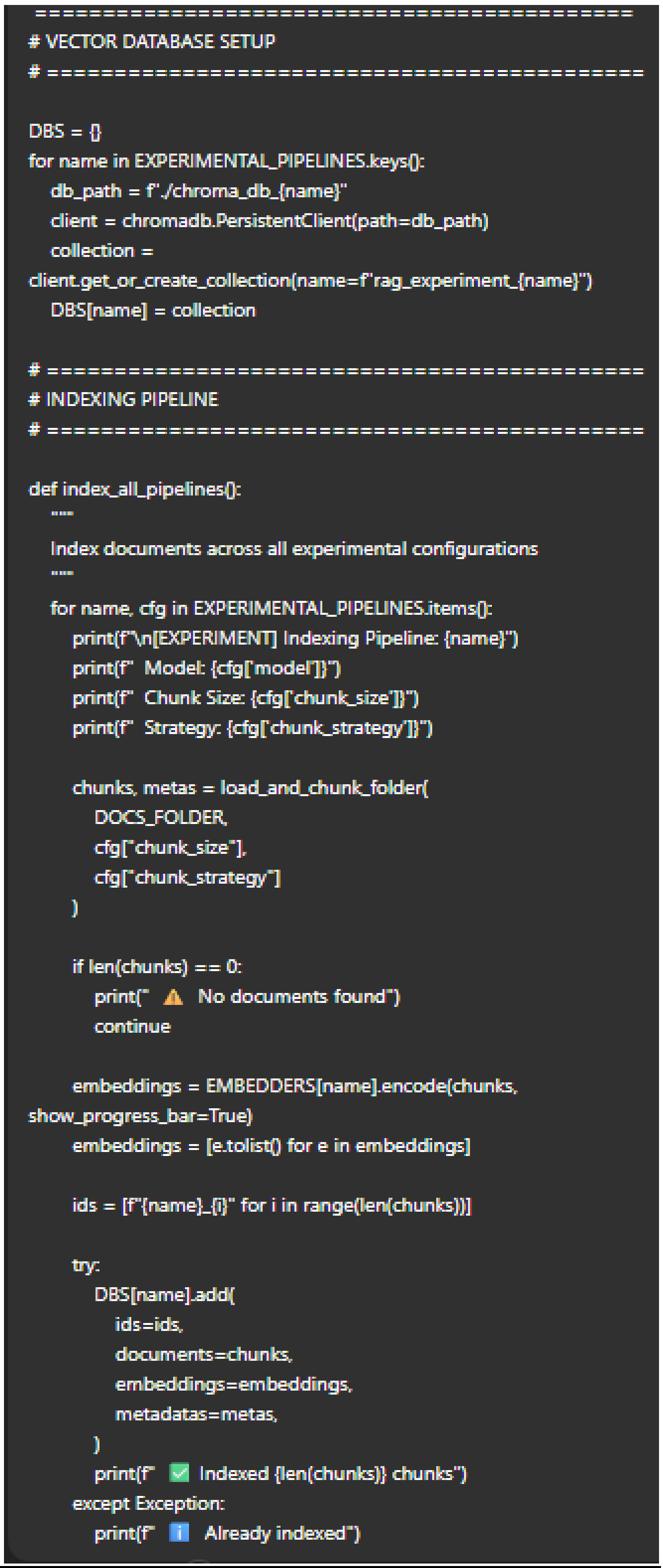
- Reload model thousands of times
- Kill performance
- Waste memory

✅ **Correct approach (what you did)**

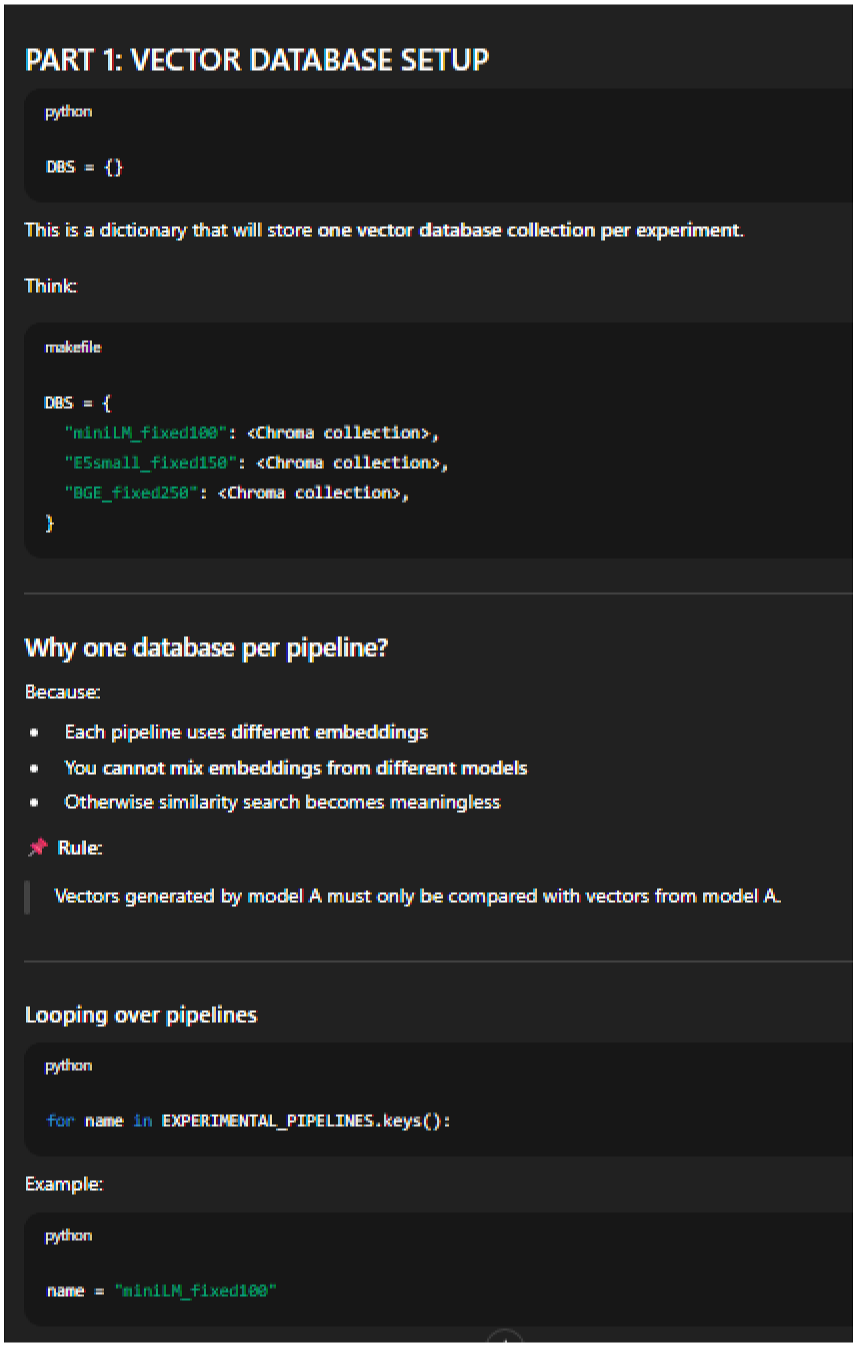
```python
Load model ONCE
Reuse it for all chunks
```

📌 This is production-grade design.

VECTOR DATABASE:



```
 ============================================
# VECTOR DATABASE SETUP
# ============================================

DBS = {}
for name in EXPERIMENTAL_PIPELINES.keys():
    db_path = f"./chroma_db_{name}"
    client = chromadb.PersistentClient(path=db_path)
    collection =
client.get_or_create_collection(name=f"rag_experiment_{name}")
    DBS[name] = collection

# ============================================
# INDEXING PIPELINE
# ============================================

def index_all_pipelines():
    """
    Index documents across all experimental configurations
    """
    for name, cfg in EXPERIMENTAL_PIPELINES.items():
        print(f"\n[EXPERIMENT] Indexing Pipeline: {name}")
        print(f"  Model: {cfg['model']}")
        print(f"  Chunk Size: {cfg['chunk_size']}")
        print(f"  Strategy: {cfg['chunk_strategy']}")

        chunks, metas = load_and_chunk_folder(
            DOCS_FOLDER,
            cfg["chunk_size"],
            cfg["chunk_strategy"]
        )

        if len(chunks) == 0:
            print("  ⚠  No documents found")
            continue

        embeddings = EMBEDDERS[name].encode(chunks,
show_progress_bar=True)
        embeddings = [e.tolist() for e in embeddings]

        ids = [f"{name}_{i}" for i in range(len(chunks))]

        try:
            DBS[name].add(
                ids=ids,
                documents=chunks,
                embeddings=embeddings,
                metadatas=metas,
            )
            print(f"  ✅  Indexed {len(chunks)} chunks")
        except Exception:
            print(f"  ℹ️  Already indexed")
```



PART 1: VECTOR DATABASE SETUP

python

```python
DBS = {}
```

This is a dictionary that will store one vector database collection per experiment.

Think:

makefile

```makefile
DBS = {
  "miniLM_fixed100": <Chroma collection>,
  "E5small_fixed150": <Chroma collection>,
  "BGE_fixed250": <Chroma collection>,
}
```

Why one database per pipeline?

Because:

- Each pipeline uses different embeddings
- You cannot mix embeddings from different models
- Otherwise similarity search becomes meaningless

📌 Rule:

> Vectors generated by model A must only be compared with vectors from model A.

Looping over pipelines

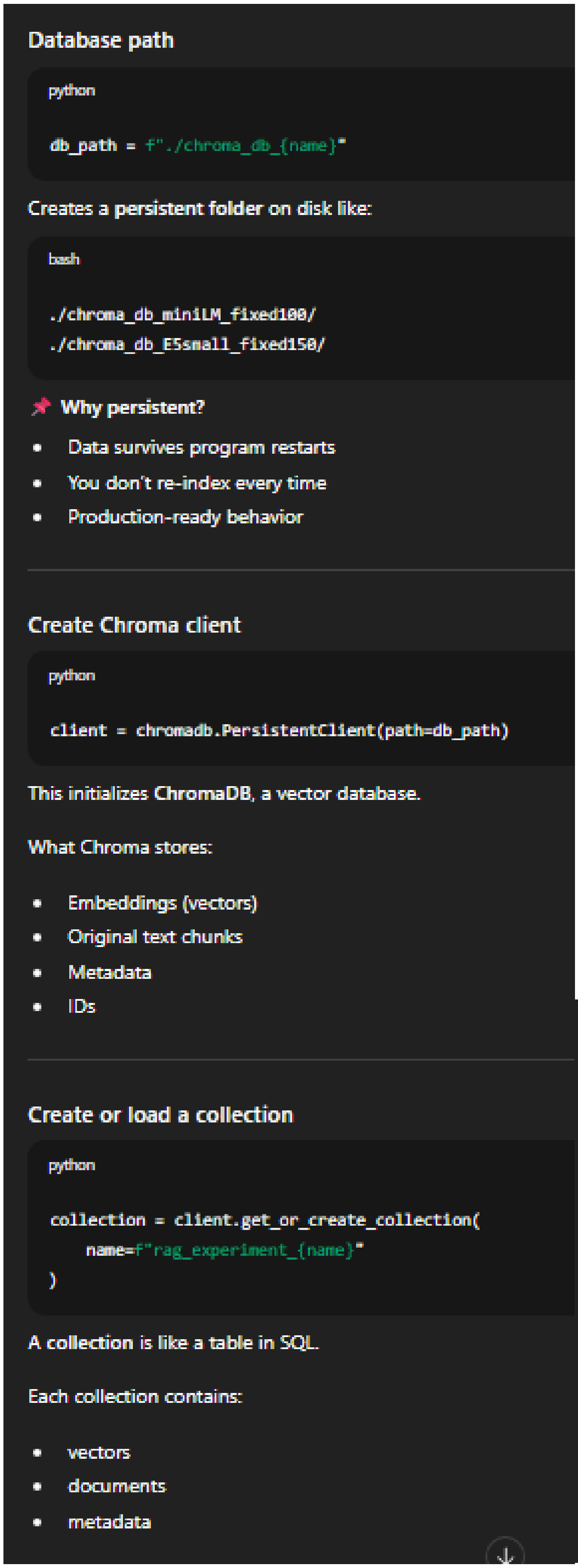
python

```python
for name in EXPERIMENTAL_PIPELINES.keys():
```

Example:

python

```python
name = "miniLM_fixed100"
```



Database path

python

```python
db_path = f"./chroma_db_{name}"
```

Creates a persistent folder on disk like:

bash

```bash
./chroma_db_miniLM_fixed100/
./chroma_db_E5small_fixed150/
```

📌 **Why persistent?**

- Data survives program restarts
- You don't re-index every time
- Production-ready behavior

Create Chroma client

python

```python
client = chromadb.PersistentClient(path=db_path)
```

This initializes ChromaDB, a vector database.

What Chroma stores:

- Embeddings (vectors)
- Original text chunks
- Metadata
- IDs

Create or load a collection

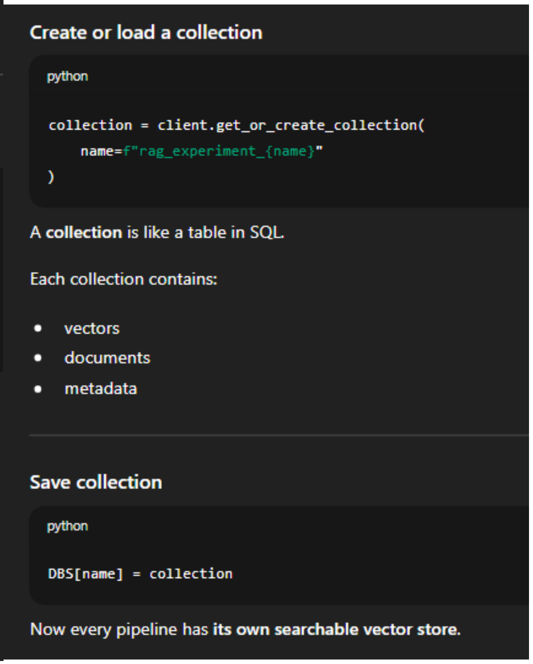
python

```python
collection = client.get_or_create_collection(
    name=f"rag_experiment_{name}"
)
```

A collection is like a table in SQL.

Each collection contains:

- vectors
- documents
- metadata



Create or load a collection

python

```python
collection = client.get_or_create_collection(
    name=f"rag_experiment_{name}"
)
```

A **collection** is like a table in SQL.

Each collection contains:

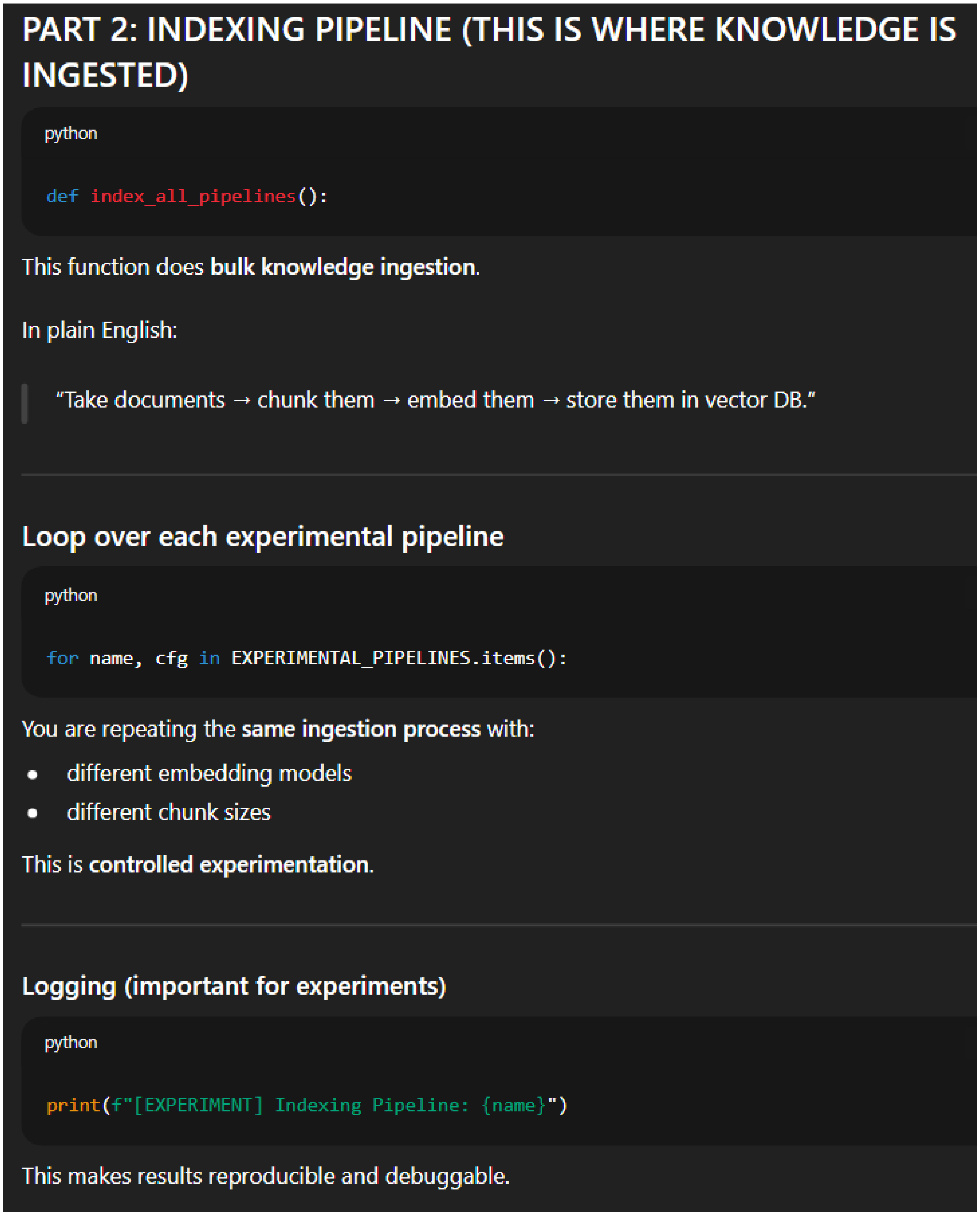
- vectors
- documents
- metadata

Save collection

python

```python
DBS[name] = collection
```

Now every pipeline has **its own searchable vector store.**



PART 2: INDEXING PIPELINE (THIS IS WHERE KNOWLEDGE IS INGESTED)

python

```python
def index_all_pipelines():
```

This function does **bulk knowledge ingestion**.

In plain English:

> "Take documents → chunk them → embed them → store them in vector DB."

Loop over each experimental pipeline

python

```python
for name, cfg in EXPERIMENTAL_PIPELINES.items():
```

You are repeating the **same ingestion process** with:

- different embedding models
- different chunk sizes

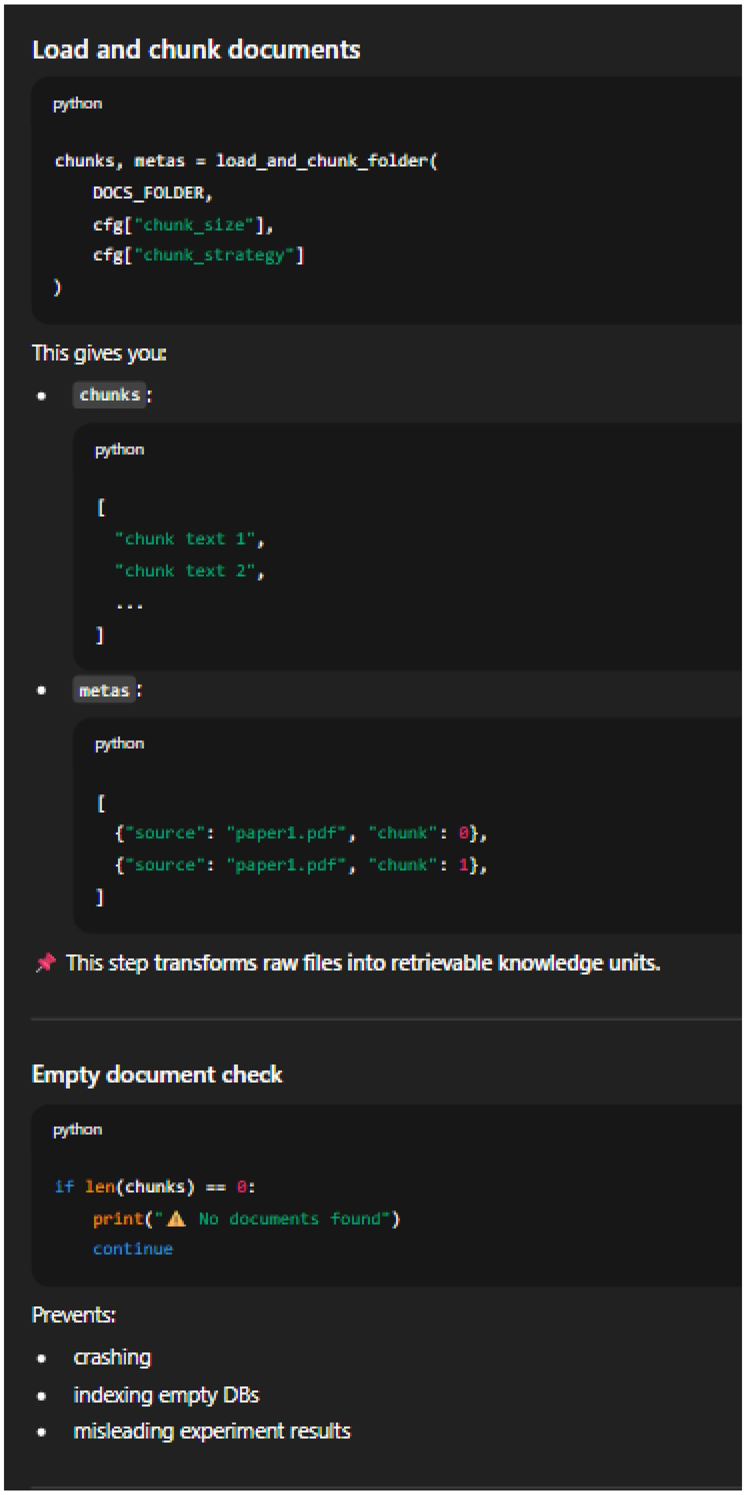
This is **controlled experimentation.**

Logging (important for experiments)

python

```python
print(f"[EXPERIMENT] Indexing Pipeline: {name}")
```

This makes results reproducible and debuggable.



Load and chunk documents

python

```python
chunks, metas = load_and_chunk_folder(
    DOCS_FOLDER,
    cfg["chunk_size"],
    cfg["chunk_strategy"]
)
```

This gives you:

- `chunks`:

  python

  ```python
  [
    "chunk text 1",
    "chunk text 2",
    ...
  ]
  ```

- `metas`:

  python

  ```python
  [
    {"source": "paper1.pdf", "chunk": 0},
    {"source": "paper1.pdf", "chunk": 1},
  ]
  ```

📌 This step transforms raw files into retrievable knowledge units.

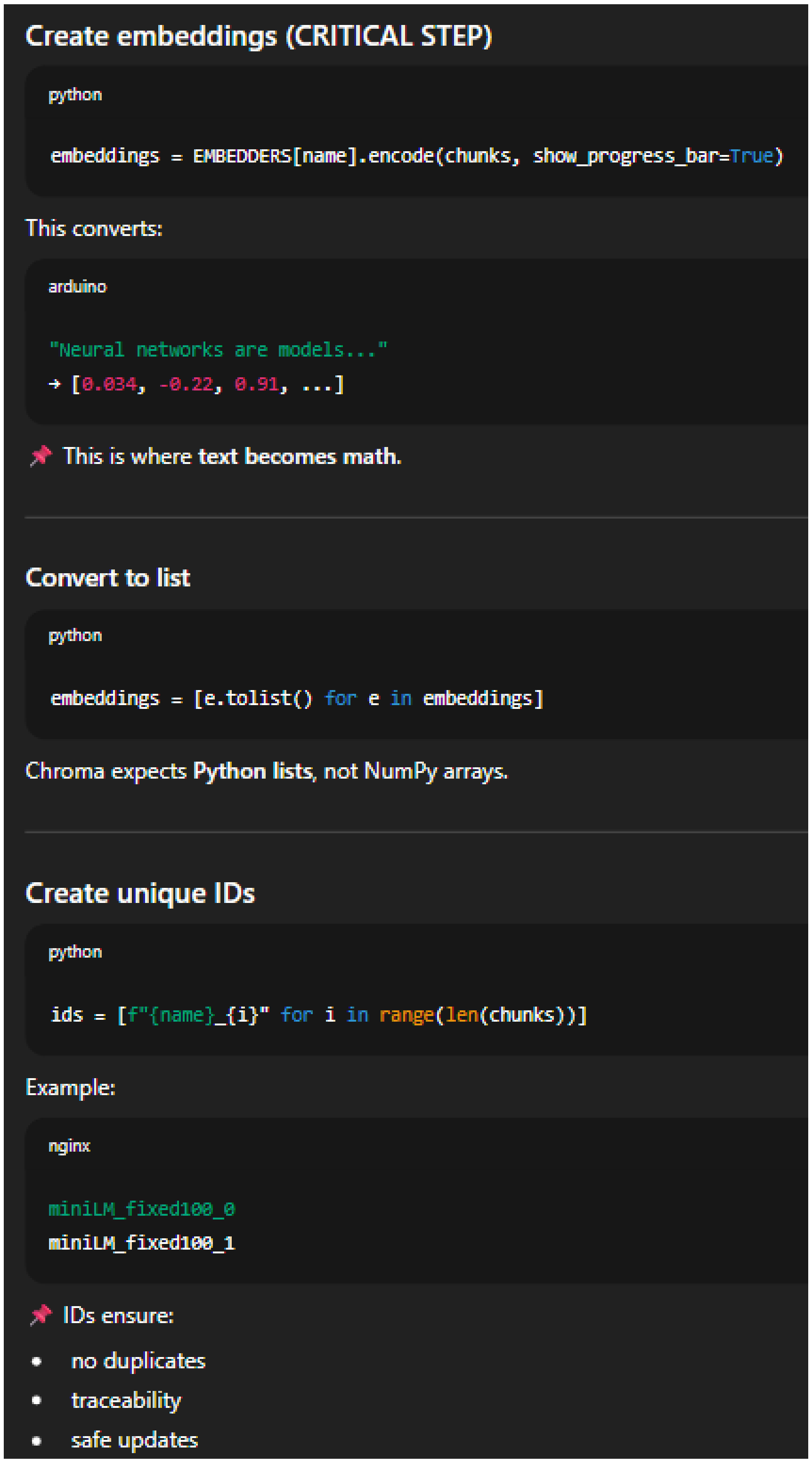
Empty document check

python

```python
if len(chunks) == 0:
    print("⚠ No documents found")
    continue
```

Prevents:

- crashing
- indexing empty DBs
- misleading experiment results



Create embeddings (CRITICAL STEP)

```python
embeddings = EMBEDDERS[name].encode(chunks, show_progress_bar=True)
```

This converts:

```arduino
"Neural networks are models..."
→ [0.034, -0.22, 0.91, ...]
```

📌 This is where **text becomes math.**

Convert to list

```python
embeddings = [e.tolist() for e in embeddings]
```

Chroma expects **Python lists**, not NumPy arrays.

Create unique IDs

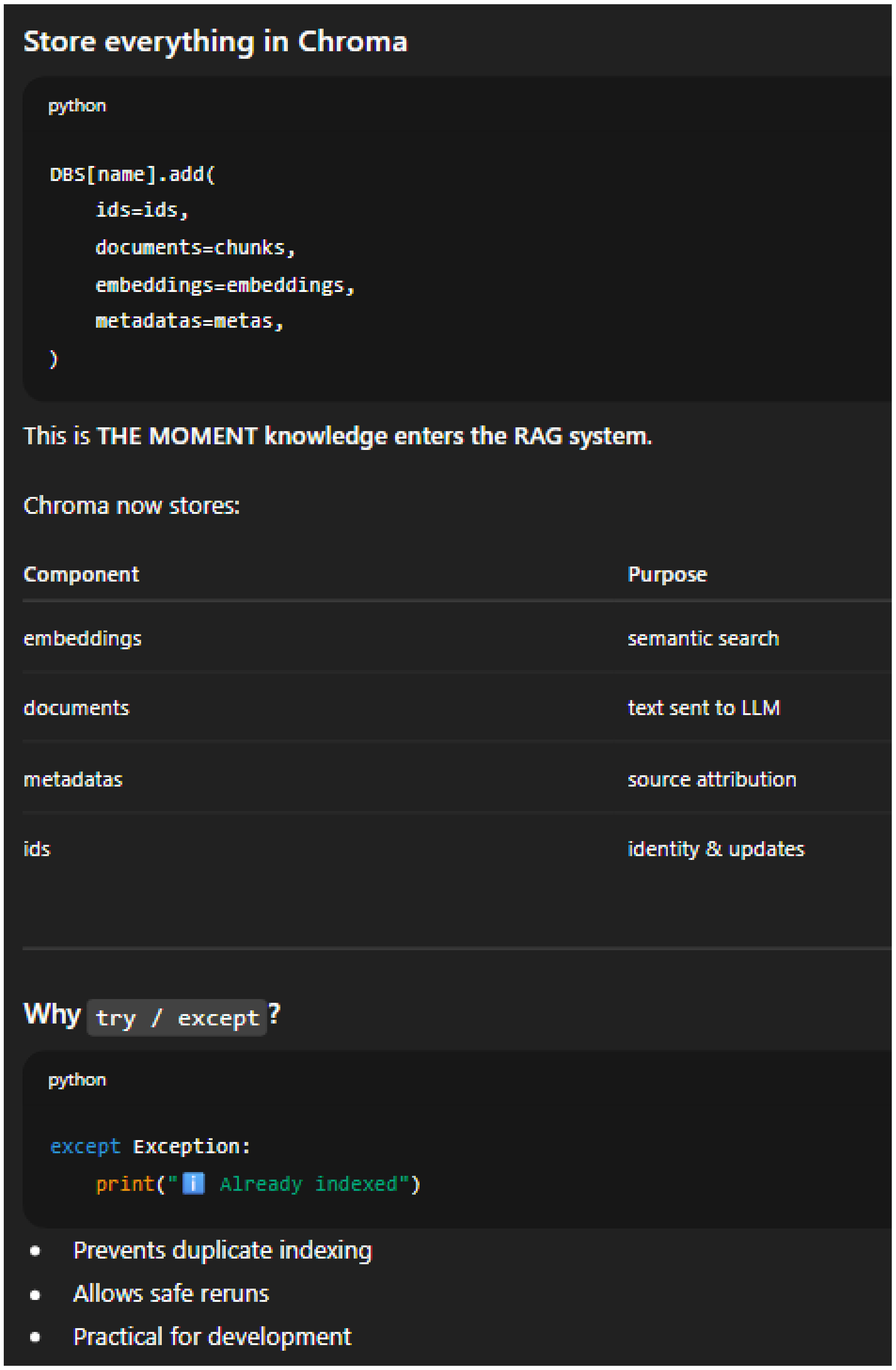
```python
ids = [f"{name}_{i}" for i in range(len(chunks))]
```

Example:

```nginx
miniLM_fixed100_0
miniLM_fixed100_1
```

📌 IDs ensure:

- no duplicates
- traceability
- safe updates



Store everything in Chroma

python

```python
DBS[name].add(
    ids=ids,
    documents=chunks,
    embeddings=embeddings,
    metadatas=metas,
)
```

This is **THE MOMENT knowledge enters the RAG system.**

Chroma now stores:

Component	Purpose
embeddings	semantic search
documents	text sent to LLM
metadatas	source attribution
ids	identity & updates

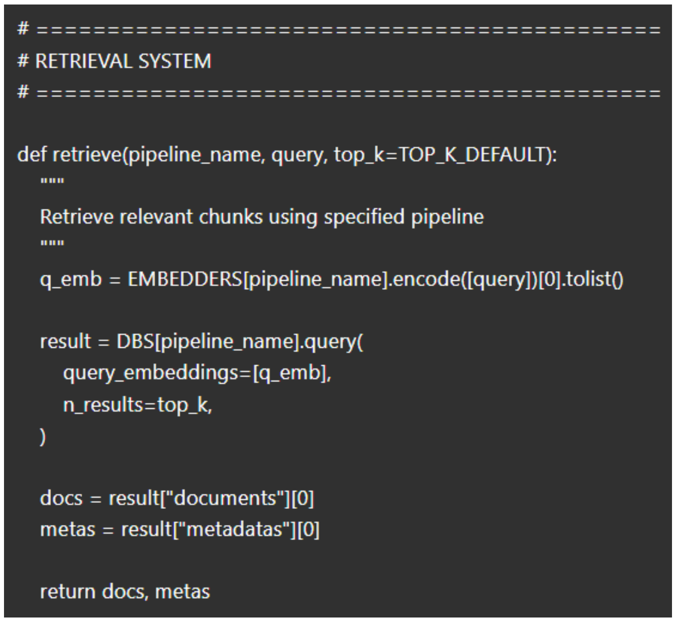
Why `try / except`?

python

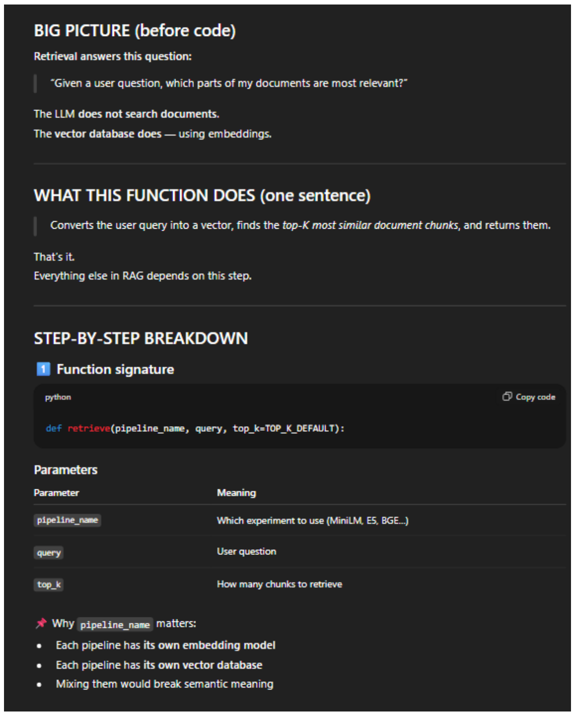
```python
except Exception:
    print("ℹ️ Already indexed")
```

- Prevents duplicate indexing
- Allows safe reruns
- Practical for development

RETRIEVAL SYSTEM



```python
# ============================================
# RETRIEVAL SYSTEM
# ============================================

def retrieve(pipeline_name, query, top_k=TOP_K_DEFAULT):
    """
    Retrieve relevant chunks using specified pipeline
    """
    q_emb = EMBEDDERS[pipeline_name].encode([query])[0].tolist()

    result = DBS[pipeline_name].query(
        query_embeddings=[q_emb],
        n_results=top_k,
    )

    docs = result["documents"][0]
    metas = result["metadatas"][0]

    return docs, metas
```



BIG PICTURE (before code)

Retrieval answers this question:

> "Given a user question, which parts of my documents are most relevant?"

The LLM **does not search documents.**
The **vector database does** — using embeddings.

WHAT THIS FUNCTION DOES (one sentence)

> Converts the user query into a vector, finds the *top-K most similar document chunks*, and returns them.

That's it.
Everything else in RAG depends on this step.

STEP-BY-STEP BREAKDOWN

1️⃣ Function signature

python

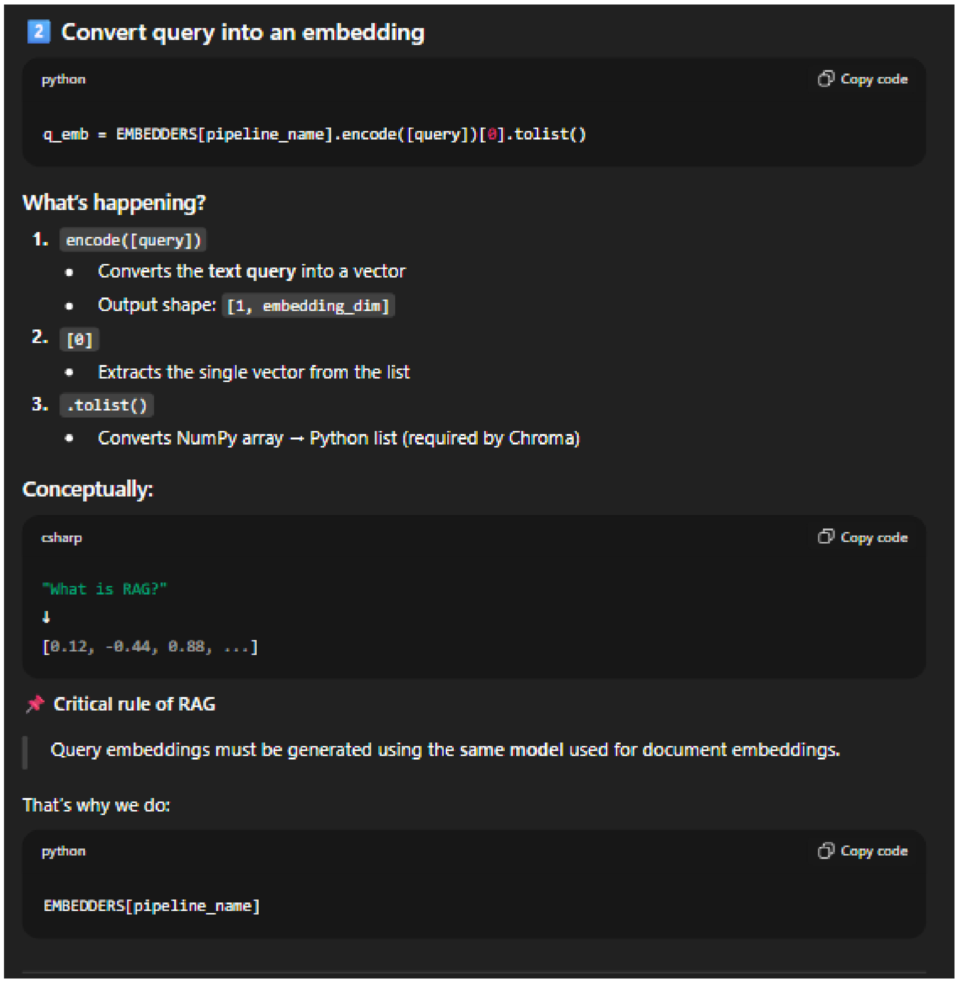
```python
def retrieve(pipeline_name, query, top_k=TOP_K_DEFAULT):
```

Parameters

Parameter	Meaning
`pipeline_name`	Which experiment to use (MiniLM, E5, BGE...)
`query`	User question
`top_k`	How many chunks to retrieve

📌 Why `pipeline_name` matters:

- Each pipeline has **its own embedding model**
- Each pipeline has **its own vector database**
- Mixing them would break semantic meaning



2️⃣ Convert query into an embedding

python

Copy code

```python
q_emb = EMBEDDERS[pipeline_name].encode([query])[0].tolist()
```

What's happening?

1. `encode([query])`
   - Converts the **text query** into a vector
   - Output shape: `[1, embedding_dim]`
2. `[0]`
   - Extracts the single vector from the list
3. `.tolist()`
   - Converts NumPy array → Python list (required by Chroma)

Conceptually:

csharp

Copy code

```csharp
"What is RAG?"
↓
[0.12, -0.44, 0.88, ...]
```

📌 **Critical rule of RAG**

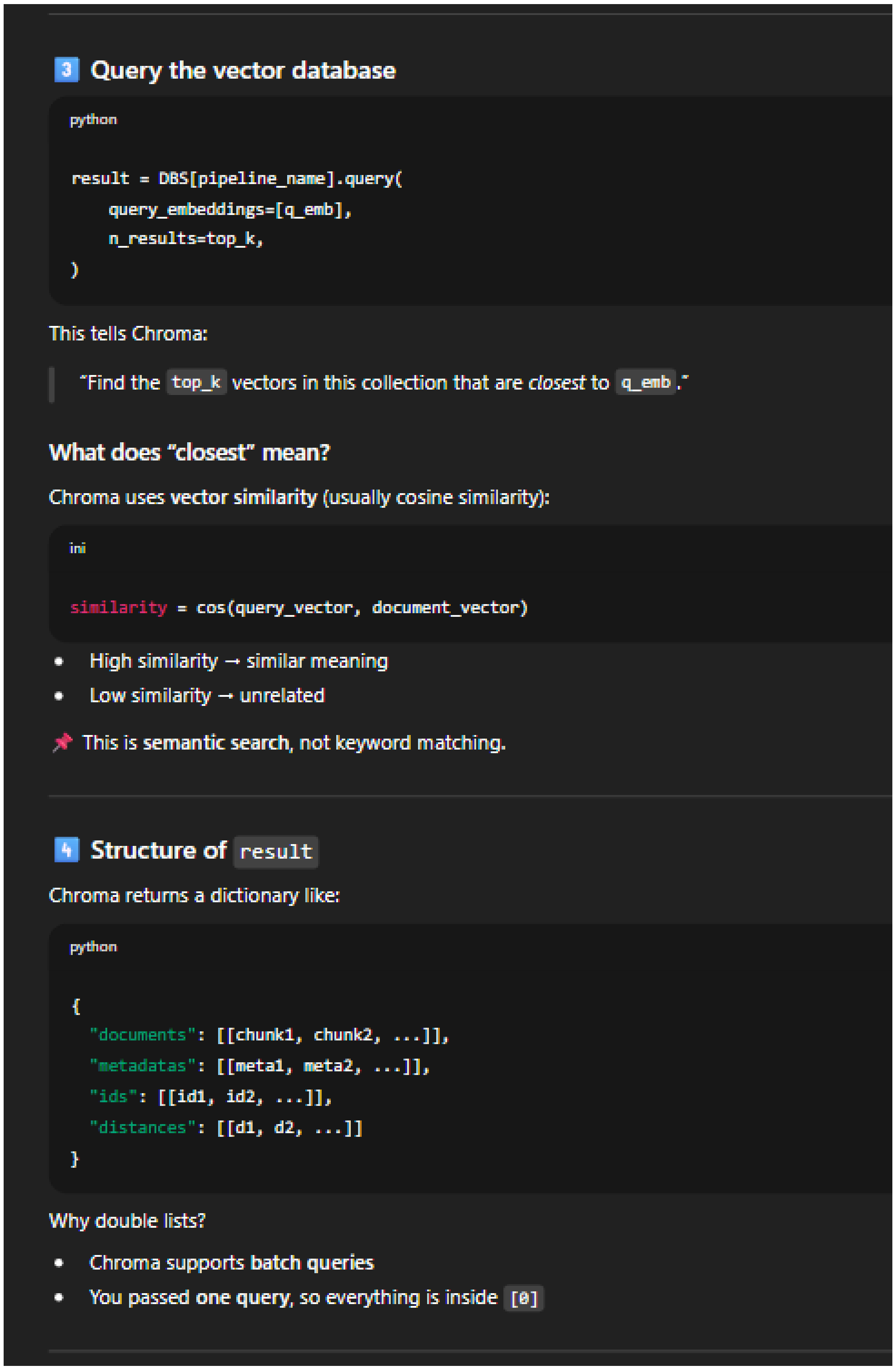
> Query embeddings must be generated using the **same model** used for document embeddings.

That's why we do:

python

Copy code

```python
EMBEDDERS[pipeline_name]
```



3️⃣ Query the vector database

python
```python
result = DBS[pipeline_name].query(
    query_embeddings=[q_emb],
    n_results=top_k,
)
```

This tells Chroma:

> "Find the `top_k` vectors in this collection that are *closest* to `q_emb`."

What does "closest" mean?

Chroma uses **vector similarity** (usually cosine similarity):

ini
```ini
similarity = cos(query_vector, document_vector)
```

- High similarity → similar meaning
- Low similarity → unrelated

📌 This is **semantic search**, not keyword matching.

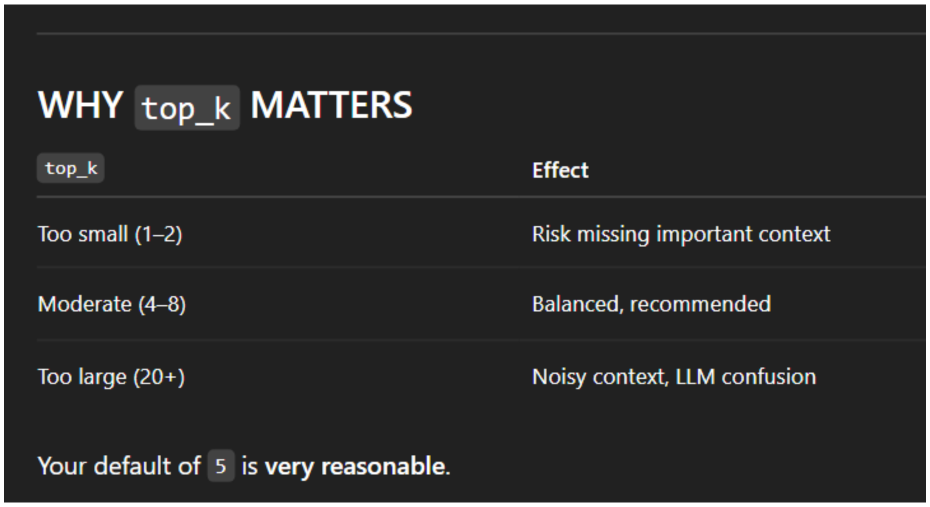
4️⃣ Structure of `result`

Chroma returns a dictionary like:

python
```python
{
  "documents": [[chunk1, chunk2, ...]],
  "metadatas": [[meta1, meta2, ...]],
  "ids": [[id1, id2, ...]],
  "distances": [[d1, d2, ...]]
}
```

Why double lists?

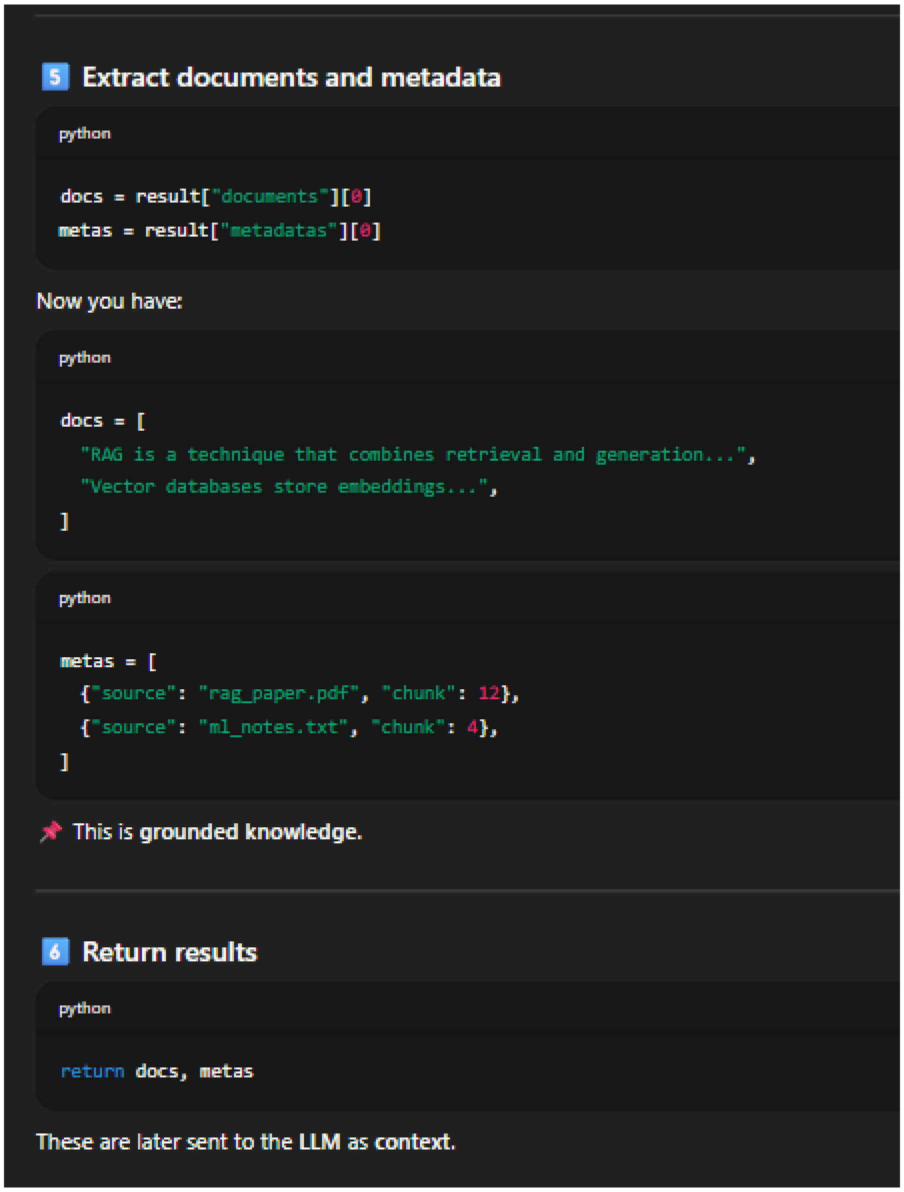
- Chroma supports **batch queries**
- You passed **one query**, so everything is inside `[0]`



WHY `top_k` MATTERS

`top_k`	Effect
Too small (1–2)	Risk missing important context
Moderate (4–8)	Balanced, recommended
Too large (20+)	Noisy context, LLM confusion

Your default of `5` is **very reasonable**.



5️⃣ Extract documents and metadata

```python
docs = result["documents"][0]
metas = result["metadatas"][0]
```

Now you have:

```python
docs = [
  "RAG is a technique that combines retrieval and generation...",
  "Vector databases store embeddings...",
]
```

```python
metas = [
  {"source": "rag_paper.pdf", "chunk": 12},
  {"source": "ml_notes.txt", "chunk": 4},
]
```

📌 This is **grounded knowledge**.

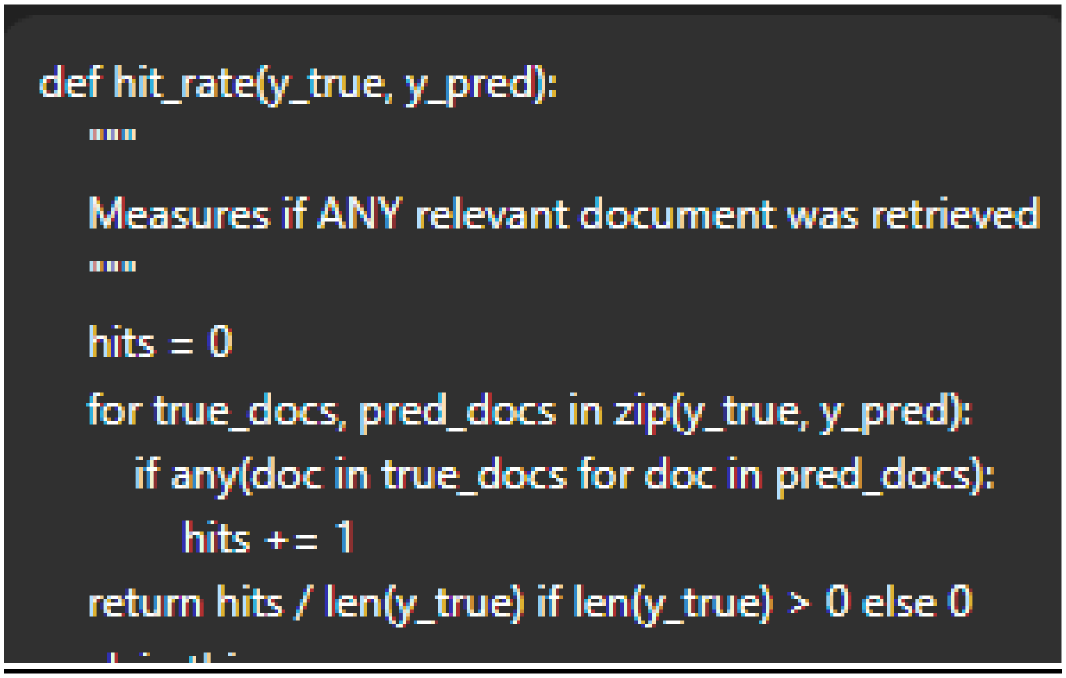
6️⃣ Return results

```python
return docs, metas
```

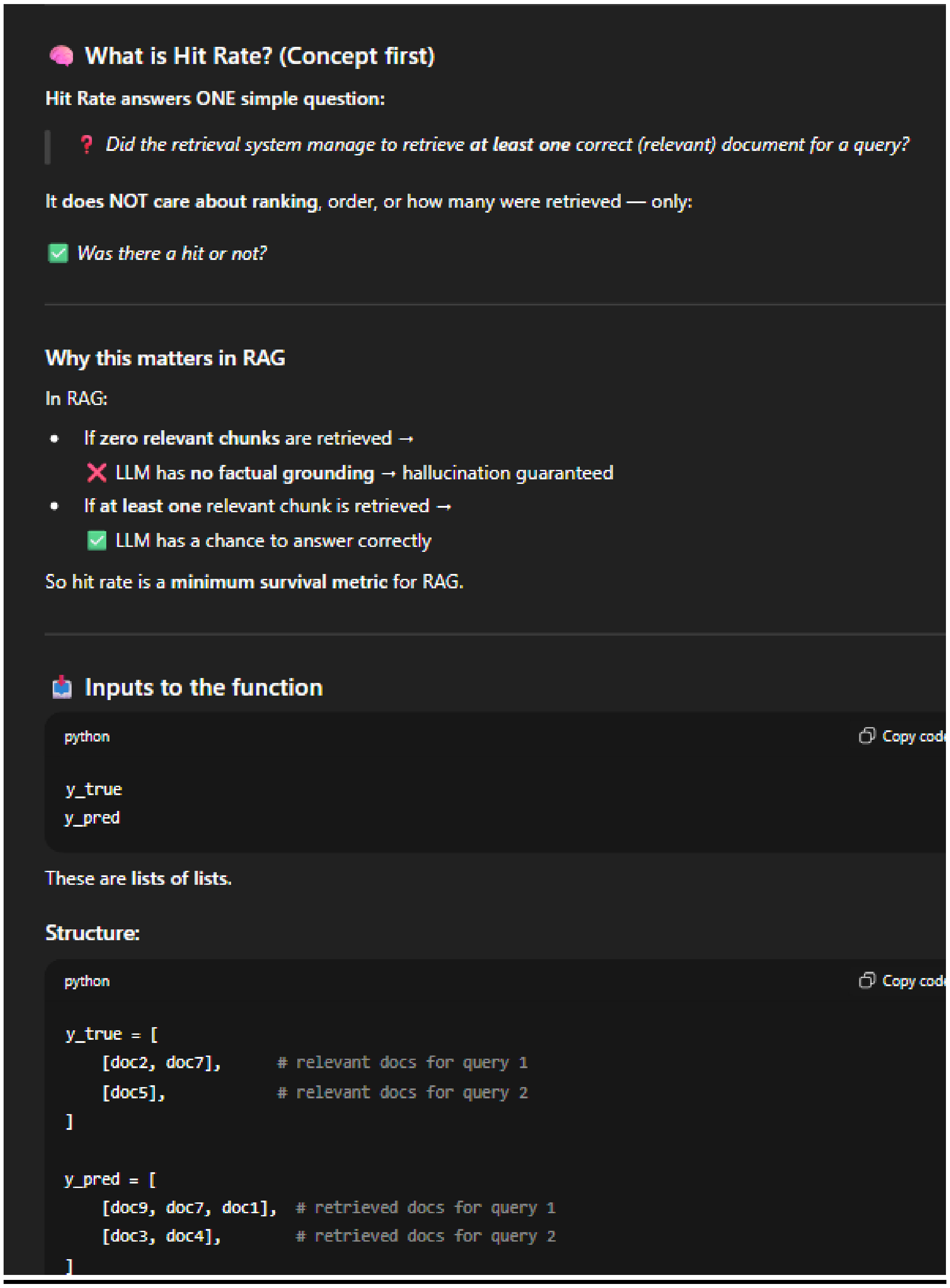
These are later sent to the LLM as context.

EVALUATION

Hit rate



```
def hit_rate(y_true, y_pred):
    """
    Measures if ANY relevant document was retrieved
    """
    hits = 0
    for true_docs, pred_docs in zip(y_true, y_pred):
        if any(doc in true_docs for doc in pred_docs):
            hits += 1
    return hits / len(y_true) if len(y_true) > 0 else 0
```



🧠 What is Hit Rate? (Concept first)

Hit Rate answers ONE simple question:

> ❓ *Did the retrieval system manage to retrieve **at least one** correct (relevant) document for a query?*

It **does NOT care about ranking**, order, or how many were retrieved — only:

✅ *Was there a hit or not?*

Why this matters in RAG

In RAG:

- If **zero relevant chunks** are retrieved →
  ❌ LLM has **no factual grounding** → hallucination guaranteed
- If **at least one** relevant chunk is retrieved →
  ✅ LLM has a chance to answer correctly

So hit rate is a **minimum survival metric** for RAG.

📥 Inputs to the function

python

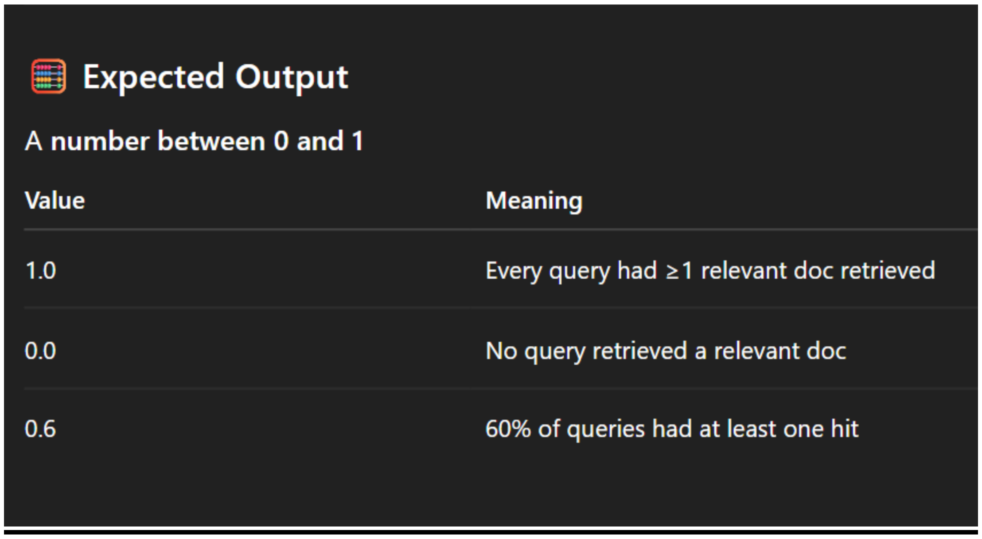
```python
y_true
y_pred
```

These are **lists of lists.**

Structure:

python

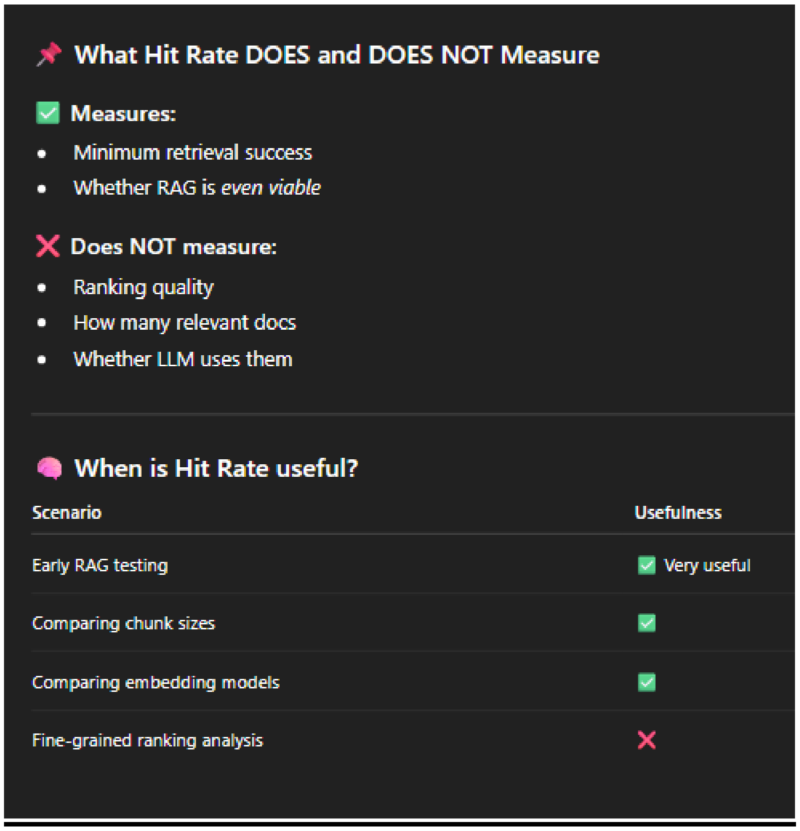
```python
y_true = [
    [doc2, doc7],      # relevant docs for query 1
    [doc5],            # relevant docs for query 2
]

y_pred = [
    [doc9, doc7, doc1],  # retrieved docs for query 1
    [doc3, doc4],        # retrieved docs for query 2
]
```



🧮 Expected Output

A number between 0 and 1

Value	Meaning
1.0	Every query had ≥1 relevant doc retrieved
0.0	No query retrieved a relevant doc
0.6	60% of queries had at least one hit



📌 What Hit Rate DOES and DOES NOT Measure

✅ **Measures:**

- Minimum retrieval success
- Whether RAG is *even viable*

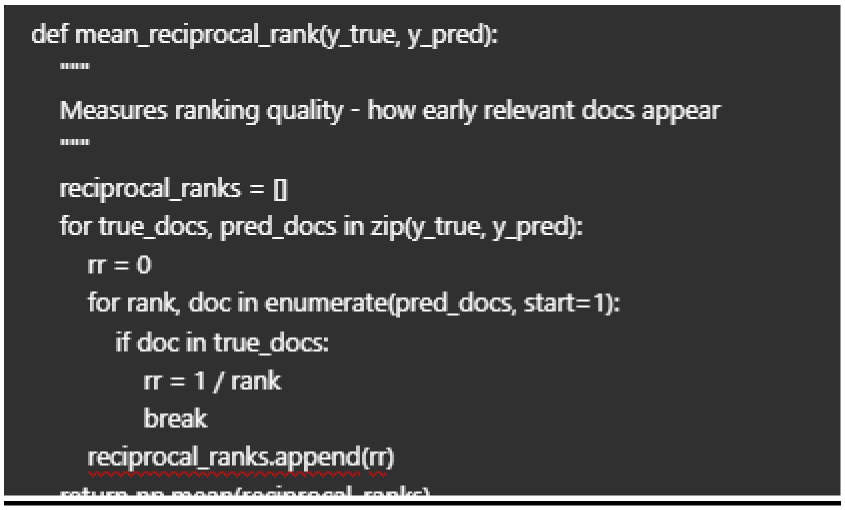
❌ **Does NOT measure:**

- Ranking quality
- How many relevant docs
- Whether LLM uses them

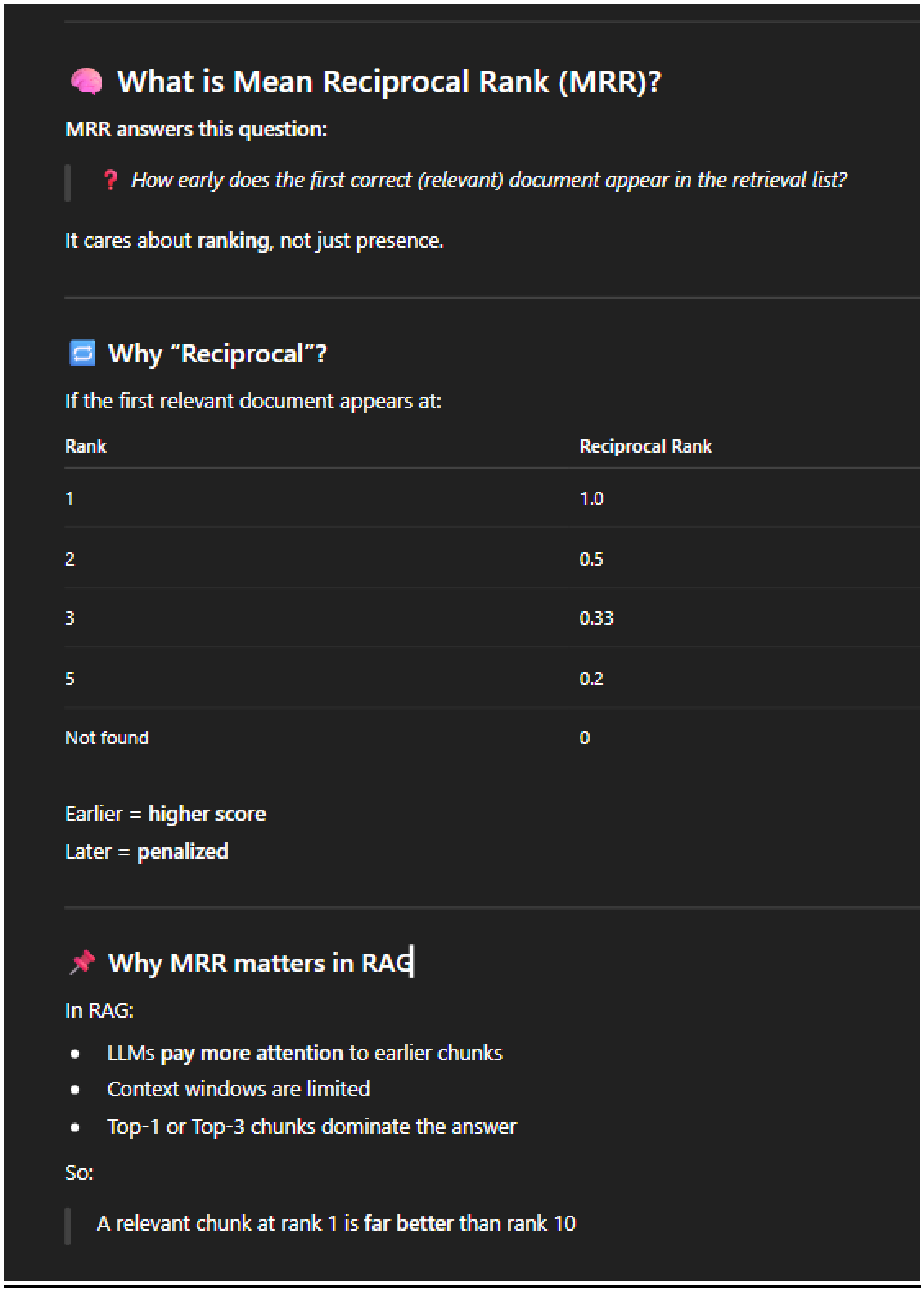
🧠 When is Hit Rate useful?

Scenario	Usefulness
Early RAG testing	✅ Very useful
Comparing chunk sizes	✅
Comparing embedding models	✅
Fine-grained ranking analysis	❌

Mean Reciprocal Rank (MRR)



```
def mean_reciprocal_rank(y_true, y_pred):
    """
    Measures ranking quality - how early relevant docs appear
    """
    reciprocal_ranks = []
    for true_docs, pred_docs in zip(y_true, y_pred):
        rr = 0
        for rank, doc in enumerate(pred_docs, start=1):
            if doc in true_docs:
                rr = 1 / rank
                break
        reciprocal_ranks.append(rr)
    return np.mean(reciprocal_ranks)
```



🧠 What is Mean Reciprocal Rank (MRR)?

MRR answers this question:

> ❓ *How early does the first correct (relevant) document appear in the retrieval list?*

It cares about **ranking**, not just presence.

🔁 Why "Reciprocal"?

If the first relevant document appears at:

Rank	Reciprocal Rank
1	1.0
2	0.5
3	0.33
5	0.2
Not found	0

Earlier = **higher score**
Later = **penalized**

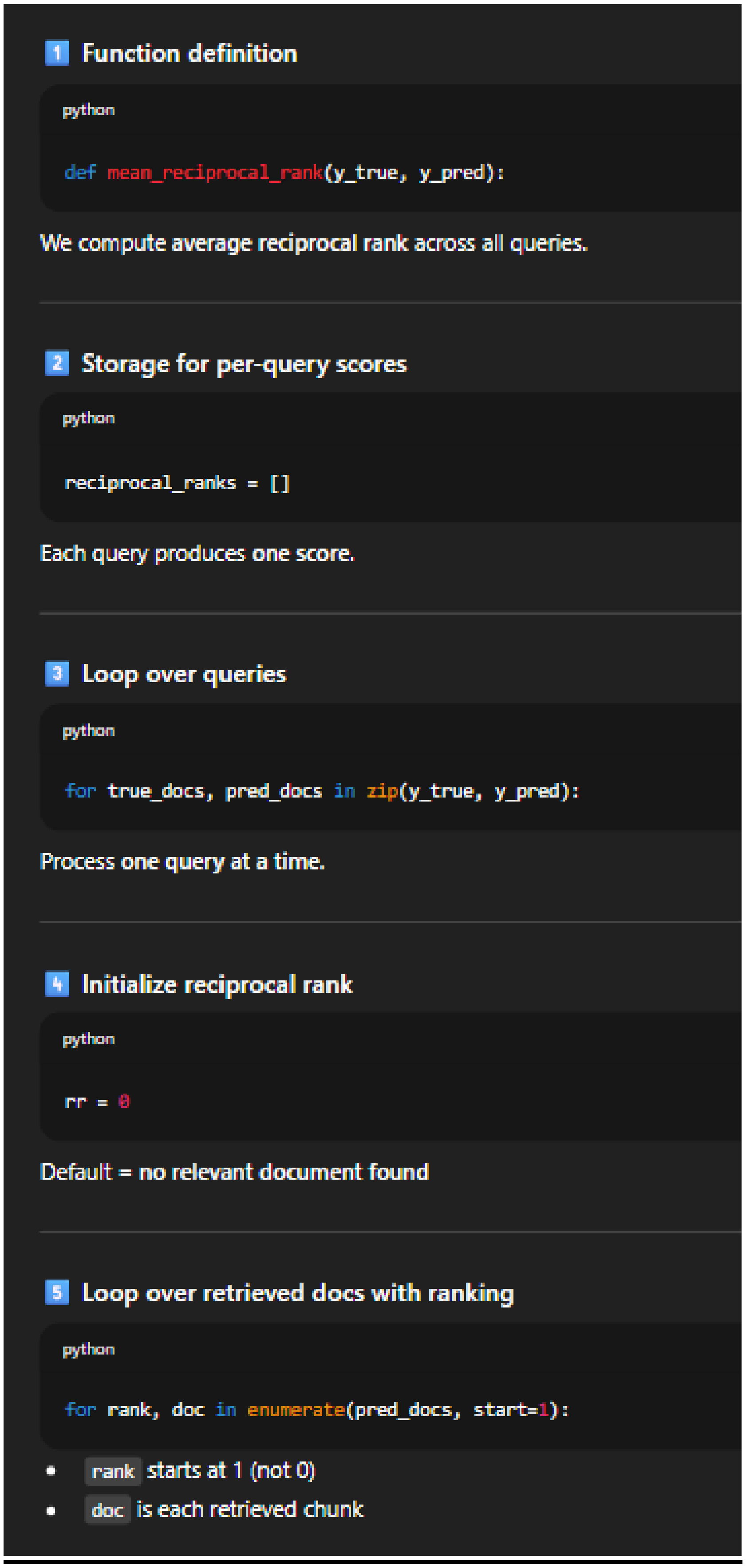
📌 Why MRR matters in RAG

In RAG:

- LLMs **pay more attention** to earlier chunks
- Context windows are limited
- Top-1 or Top-3 chunks dominate the answer

So:

> A relevant chunk at rank 1 is **far better** than rank 10



1️⃣ Function definition

python

```python
def mean_reciprocal_rank(y_true, y_pred):
```

We compute average reciprocal rank across all queries.

2️⃣ Storage for per-query scores

python

```python
reciprocal_ranks = []
```

Each query produces one score.

3️⃣ Loop over queries

python

```python
for true_docs, pred_docs in zip(y_true, y_pred):
```

Process one query at a time.

4️⃣ Initialize reciprocal rank

python

```python
rr = 0
```

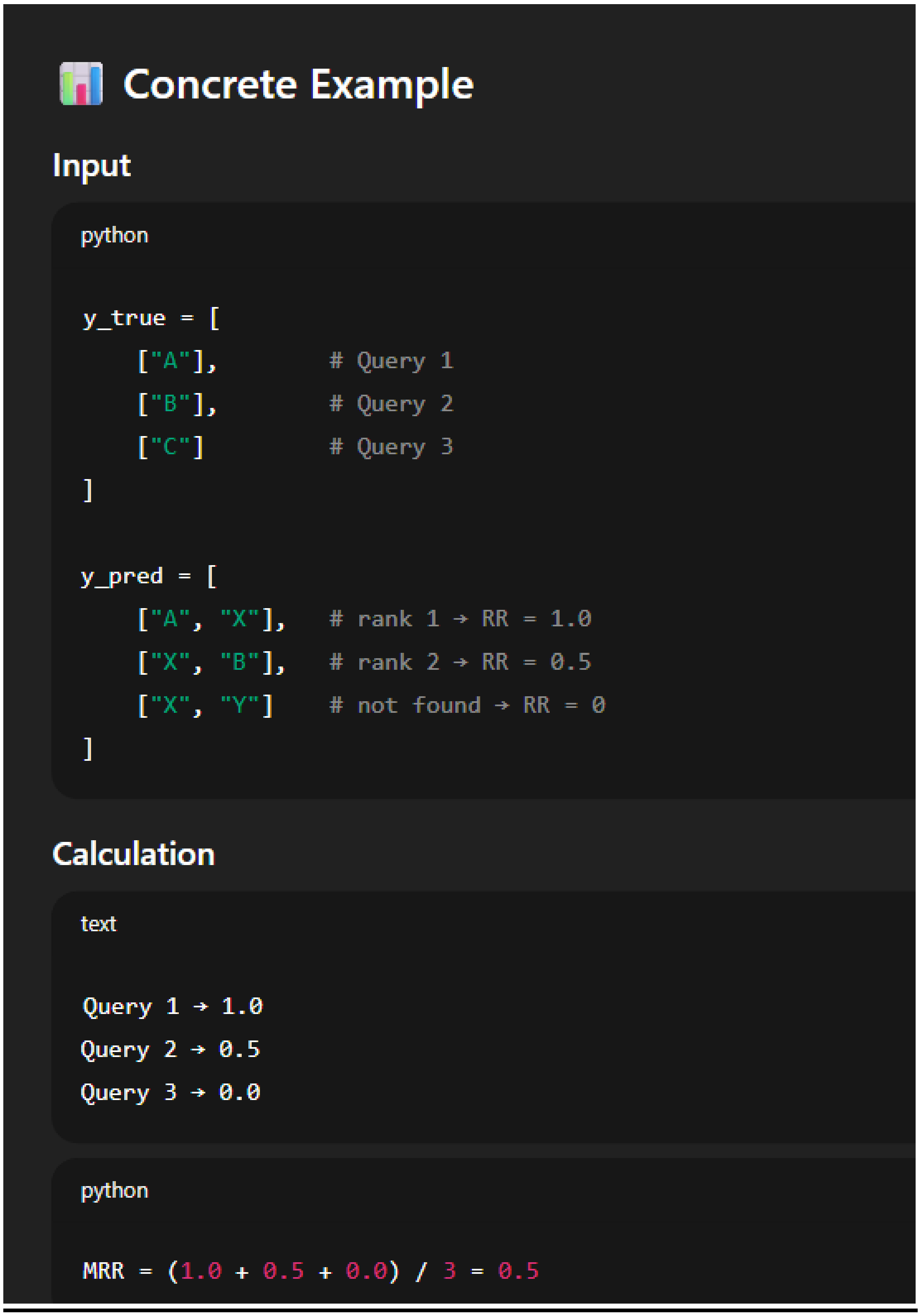
Default = no relevant document found

5️⃣ Loop over retrieved docs with ranking

python

```python
for rank, doc in enumerate(pred_docs, start=1):
```

- `rank` starts at 1 (not 0)
- `doc` is each retrieved chunk



📊 Concrete Example

Input

python

```python
y_true = [
    ["A"],        # Query 1
    ["B"],        # Query 2
    ["C"]         # Query 3
]

y_pred = [
    ["A", "X"],   # rank 1 → RR = 1.0
    ["X", "B"],   # rank 2 → RR = 0.5
    ["X", "Y"]    # not found → RR = 0
]
```

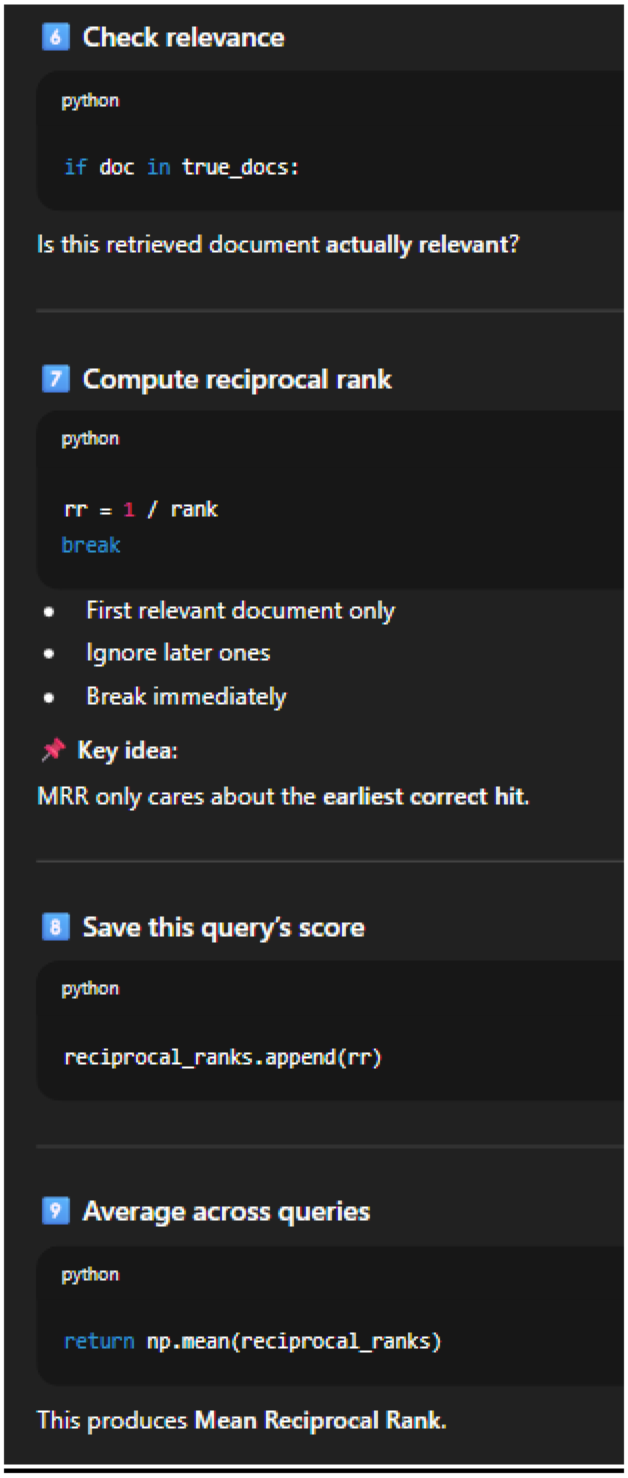
Calculation

text

```text
Query 1 → 1.0
Query 2 → 0.5
Query 3 → 0.0
```

python

```python
MRR = (1.0 + 0.5 + 0.0) / 3 = 0.5
```



6 Check relevance

python

```python
if doc in true_docs:
```

Is this retrieved document **actually relevant?**

---

7 Compute reciprocal rank

python

```python
rr = 1 / rank
break
```

- First relevant document only
- Ignore later ones
- Break immediately

📌 **Key idea:**

MRR only cares about the **earliest correct hit.**

---

8 Save this query's score

python

```python
reciprocal_ranks.append(rr)
```

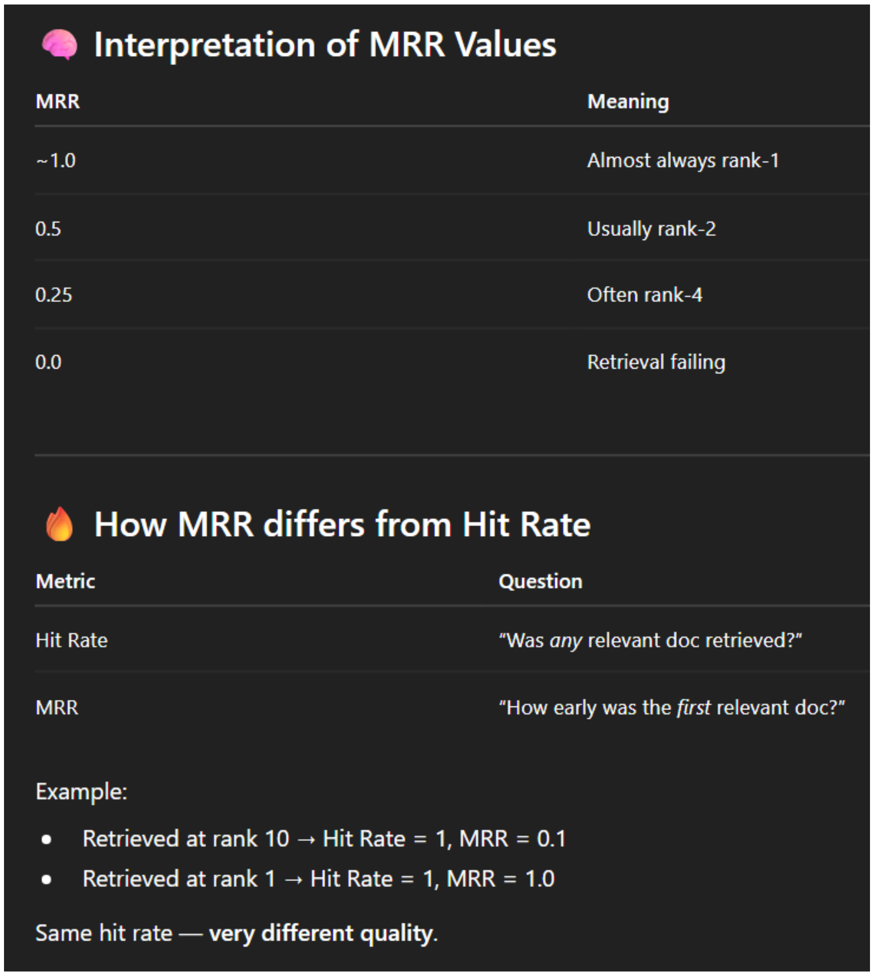
---

9 Average across queries

python

```python
return np.mean(reciprocal_ranks)
```

This produces **Mean Reciprocal Rank.**



🧠 Interpretation of MRR Values

MRR	Meaning
~1.0	Almost always rank-1
0.5	Usually rank-2
0.25	Often rank-4
0.0	Retrieval failing

🔥 How MRR differs from Hit Rate

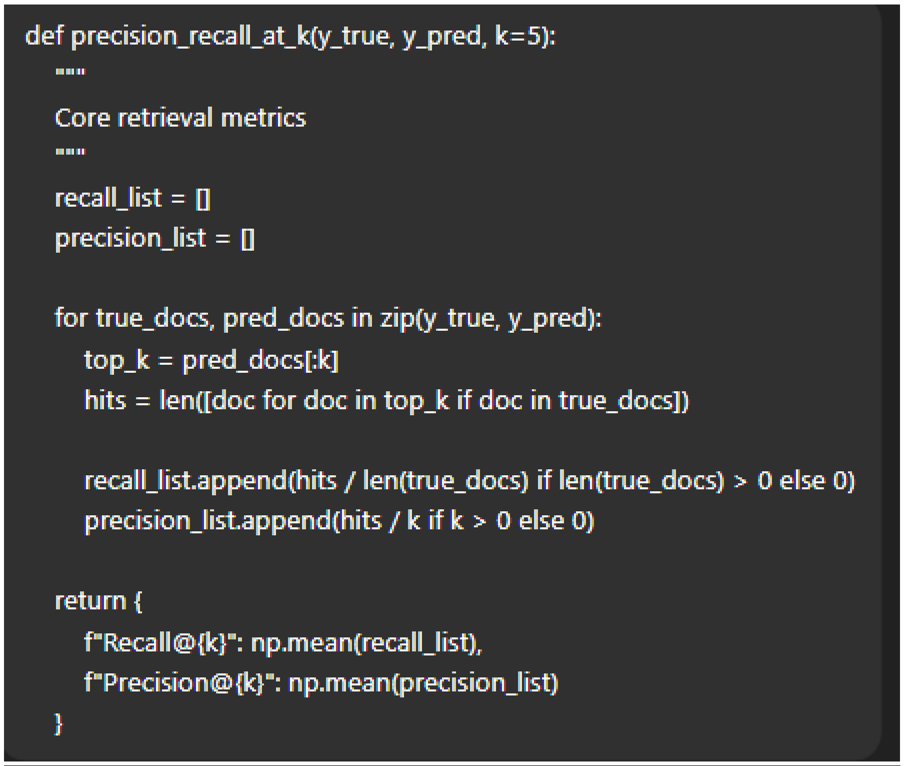
Metric	Question
Hit Rate	"Was *any* relevant doc retrieved?"
MRR	"How early was the *first* relevant doc?"

Example:

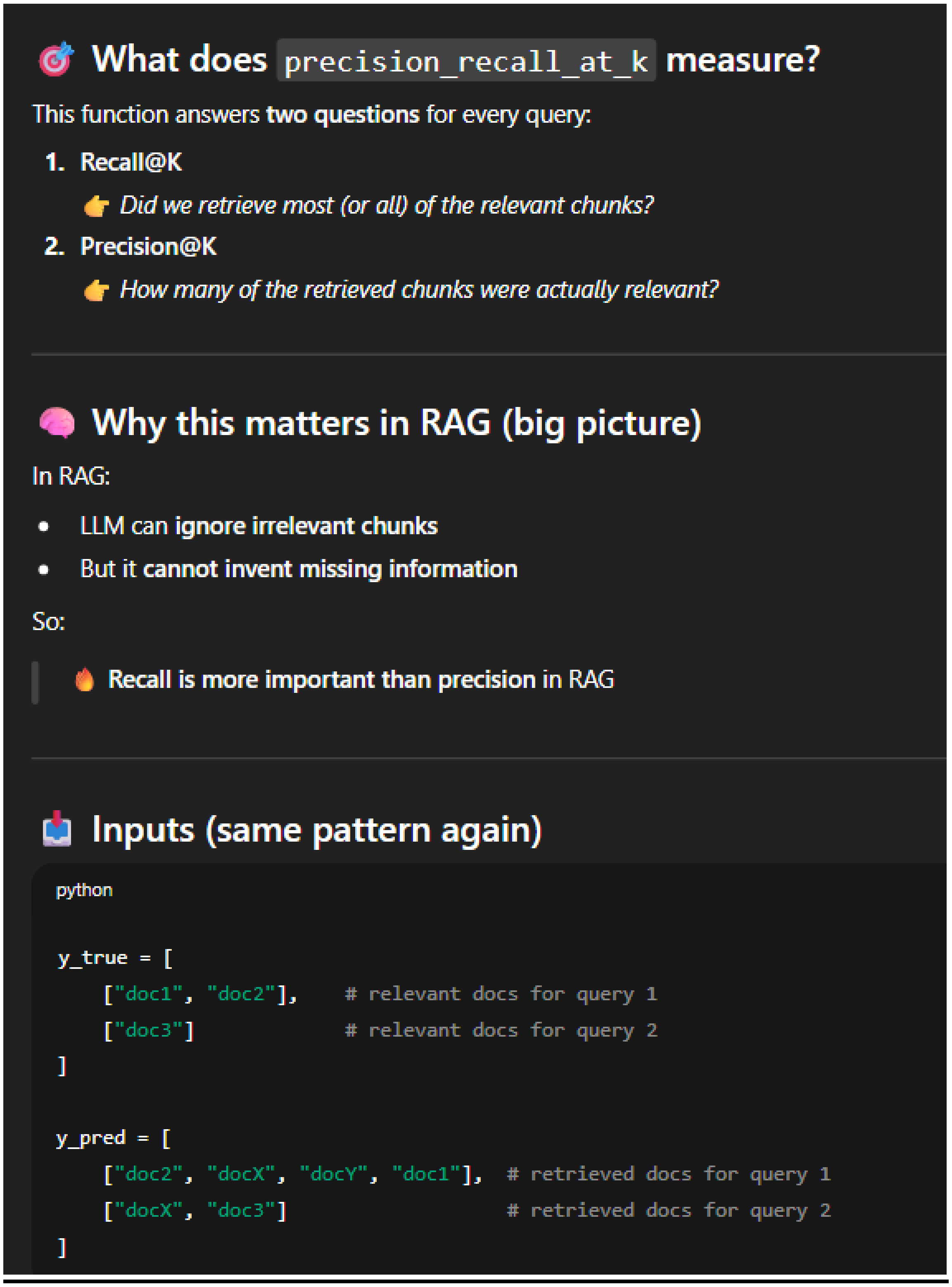
- Retrieved at rank 10 → Hit Rate = 1, MRR = 0.1
- Retrieved at rank 1 → Hit Rate = 1, MRR = 1.0

Same hit rate — **very different quality**.

precision_recall_at_k



```python
def precision_recall_at_k(y_true, y_pred, k=5):
    """
    Core retrieval metrics
    """
    recall_list = []
    precision_list = []

    for true_docs, pred_docs in zip(y_true, y_pred):
        top_k = pred_docs[:k]
        hits = len([doc for doc in top_k if doc in true_docs])

        recall_list.append(hits / len(true_docs) if len(true_docs) > 0 else 0)
        precision_list.append(hits / k if k > 0 else 0)

    return {
        f"Recall@{k}": np.mean(recall_list),
        f"Precision@{k}": np.mean(precision_list)
    }
```



🎯 What does `precision_recall_at_k` measure?

This function answers **two questions** for every query:

1. **Recall@K**
   👉 *Did we retrieve most (or all) of the relevant chunks?*
2. **Precision@K**
   👉 *How many of the retrieved chunks were actually relevant?*

🧠 Why this matters in RAG (big picture)

In RAG:

- LLM can **ignore irrelevant chunks**
- But it **cannot invent missing information**

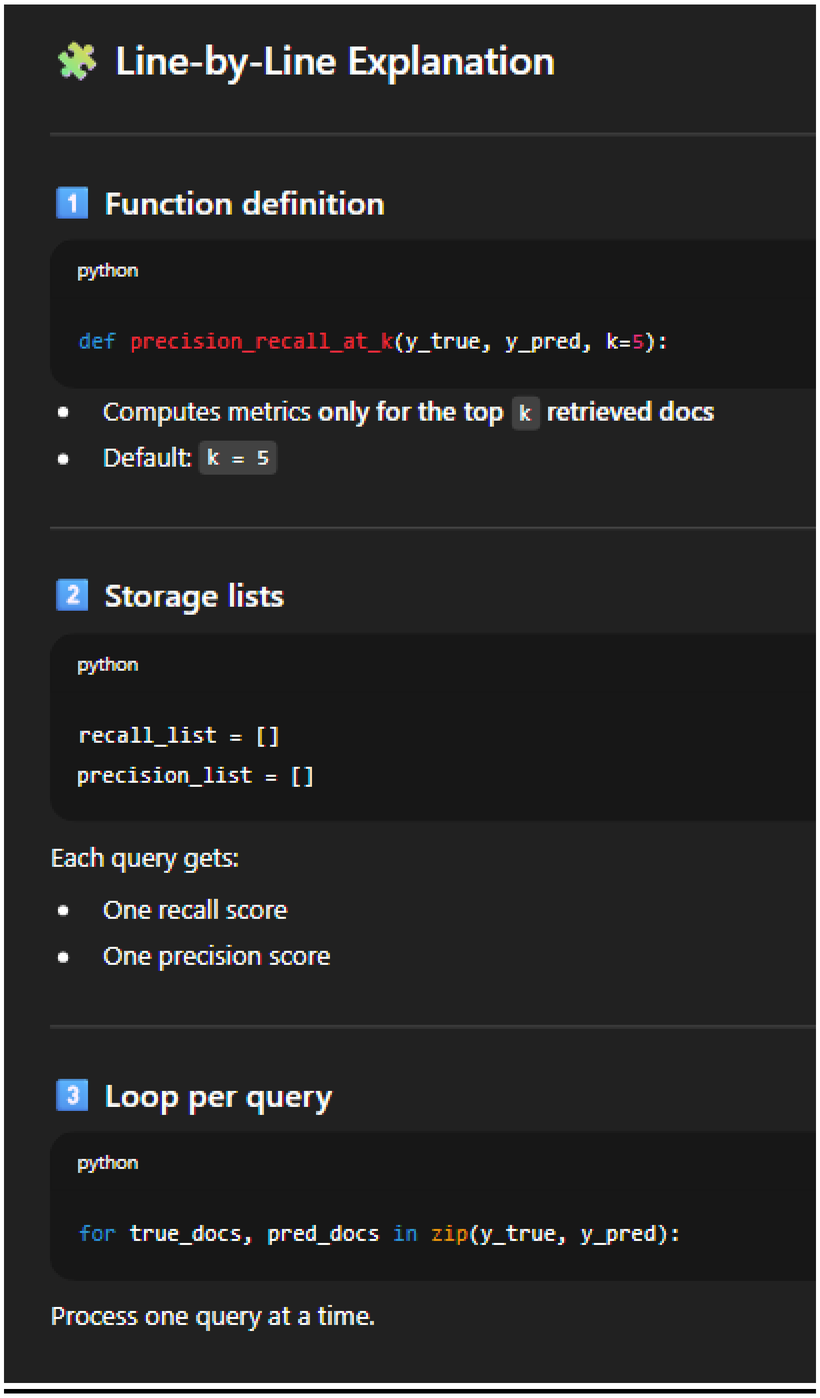
So:

> 🔥 **Recall is more important than precision** in RAG

📥 Inputs (same pattern again)

python

```python
y_true = [
    ["doc1", "doc2"],    # relevant docs for query 1
    ["doc3"]             # relevant docs for query 2
]

y_pred = [
    ["doc2", "docX", "docY", "doc1"],  # retrieved docs for query 1
    ["docX", "doc3"]                   # retrieved docs for query 2
]
```



🧩 Line-by-Line Explanation

1️⃣ Function definition

```python
def precision_recall_at_k(y_true, y_pred, k=5):
```

- Computes metrics **only for the top `k` retrieved docs**
- Default: `k = 5`

2️⃣ Storage lists

```python
recall_list = []
precision_list = []
```

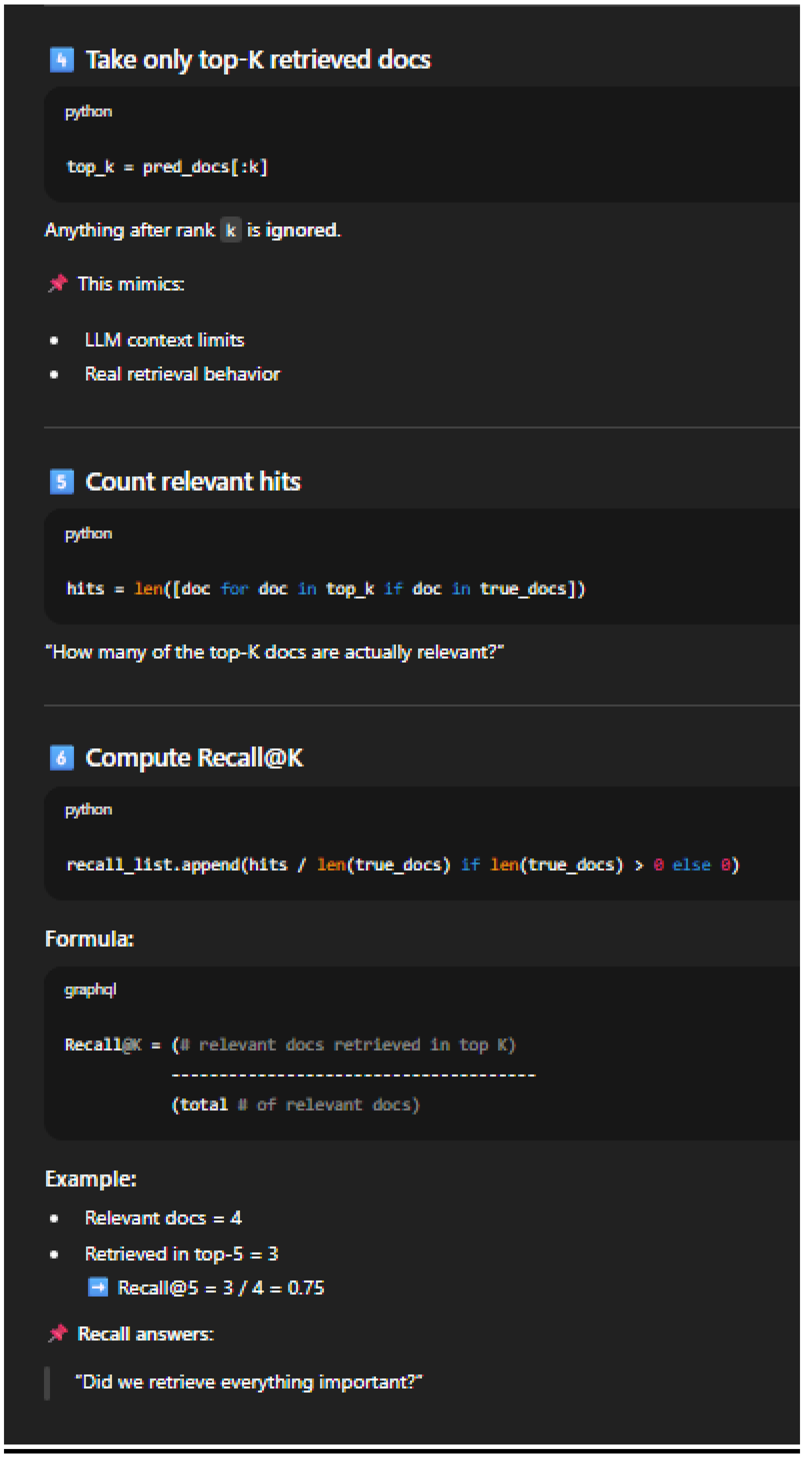
Each query gets:

- One recall score
- One precision score

3️⃣ Loop per query

```python
for true_docs, pred_docs in zip(y_true, y_pred):
```

Process one query at a time.



4️⃣ Take only top-K retrieved docs

```python
top_k = pred_docs[:k]
```

Anything after rank `k` is ignored.

📌 This mimics:

- LLM context limits
- Real retrieval behavior

5️⃣ Count relevant hits

```python
hits = len([doc for doc in top_k if doc in true_docs])
```

"How many of the top-K docs are actually relevant?"

6️⃣ Compute Recall@K

```python
recall_list.append(hits / len(true_docs) if len(true_docs) > 0 else 0)
```

Formula:

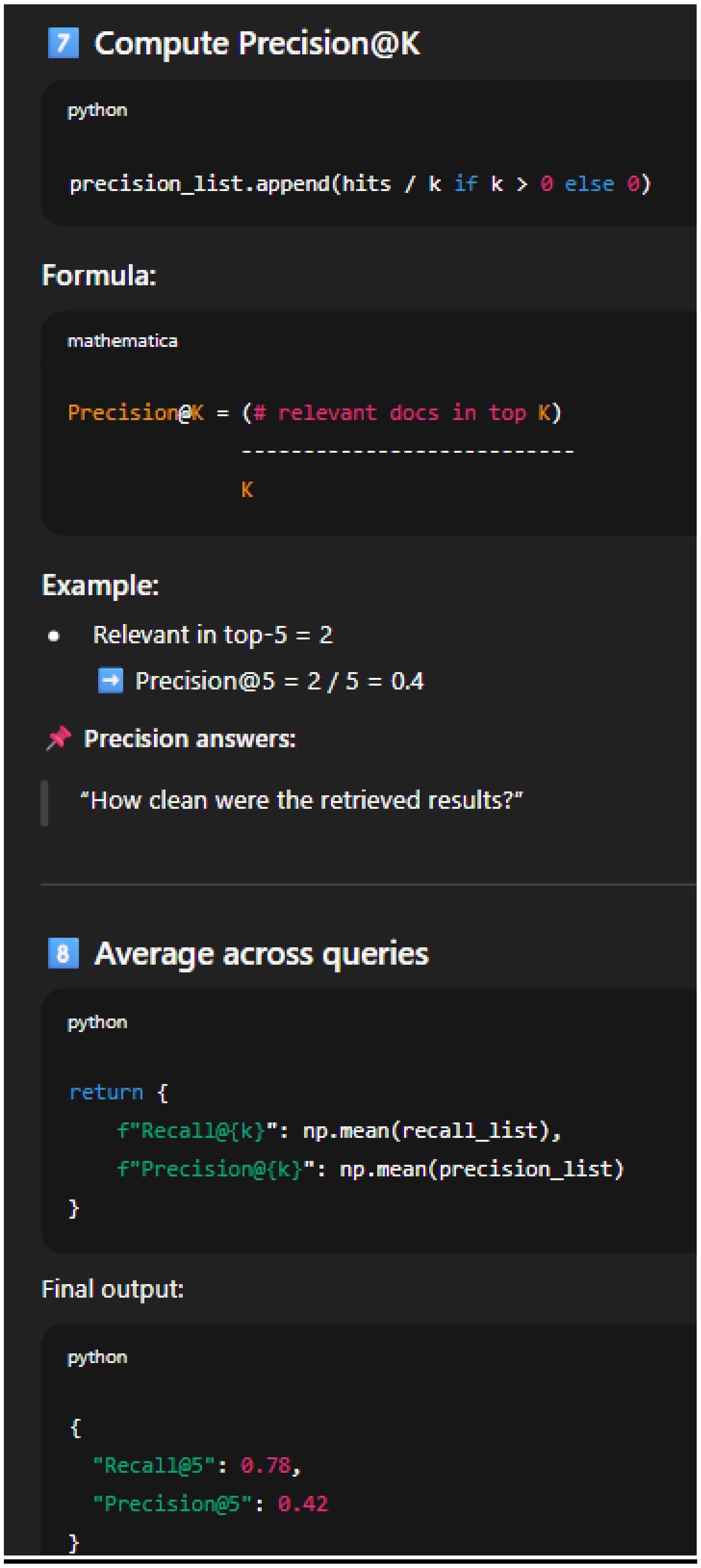
```graphql
Recall@K = (# relevant docs retrieved in top K)
           --------------------------------------
           (total # of relevant docs)
```

Example:

- Relevant docs = 4
- Retrieved in top-5 = 3
  ➡️ Recall@5 = 3 / 4 = 0.75

📌 Recall answers:

> "Did we retrieve everything important?"



7️⃣ Compute Precision@K

```python
precision_list.append(hits / k if k > 0 else 0)
```

Formula:

```mathematica
Precision@K = (# relevant docs in top K)
              ---------------------------
              K
```

Example:

- Relevant in top-5 = 2
  ➡️ Precision@5 = 2 / 5 = 0.4

📌 **Precision answers:**

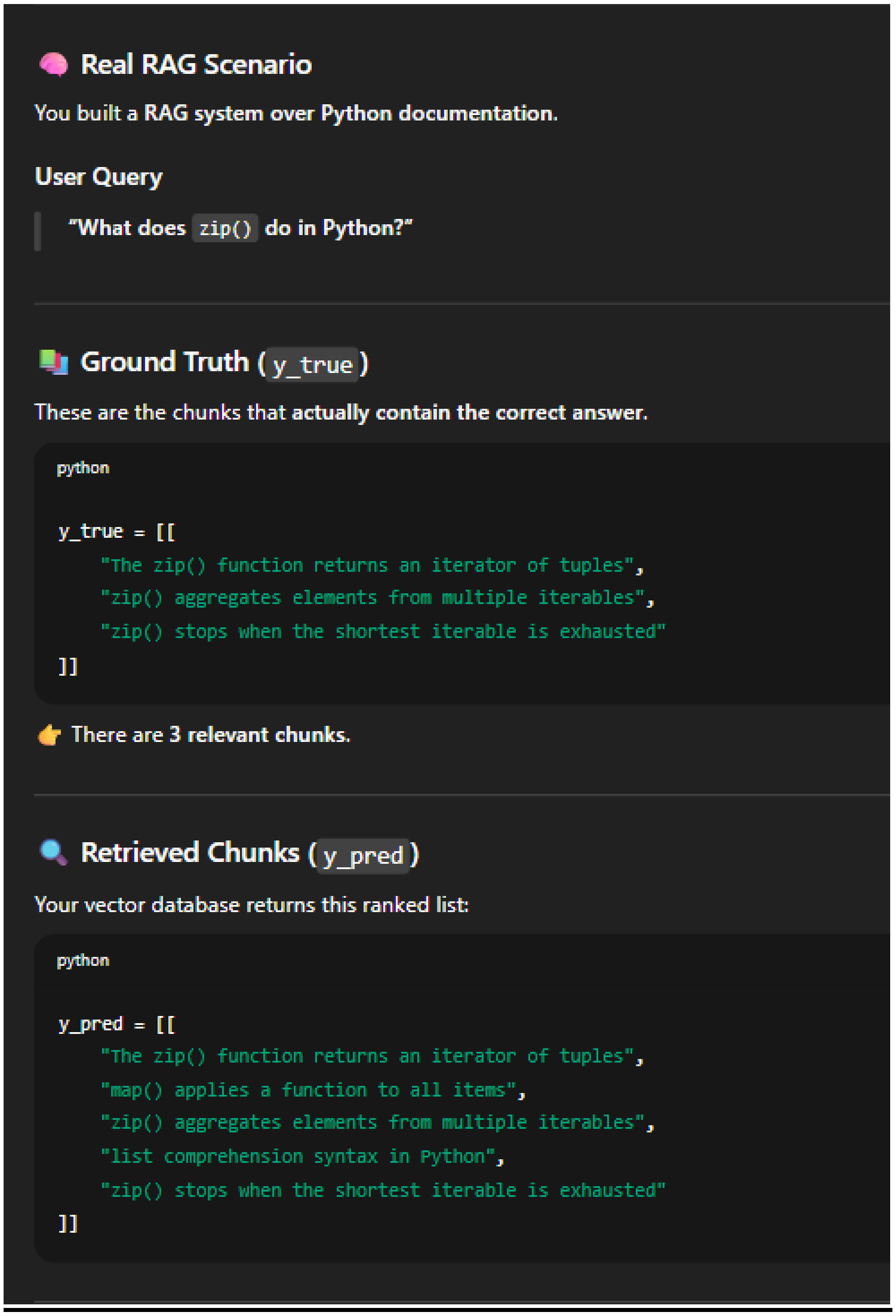
> "How clean were the retrieved results?"

8️⃣ Average across queries

```python
return {
    f"Recall@{k}": np.mean(recall_list),
    f"Precision@{k}": np.mean(precision_list)
}
```

Final output:

```python
{
  "Recall@5": 0.78,
  "Precision@5": 0.42
}
```



🧠 Real RAG Scenario

You built a **RAG system over Python documentation.**

User Query

> **"What does `zip()` do in Python?"**

📚 Ground Truth (`y_true`)

These are the chunks that **actually contain the correct answer.**

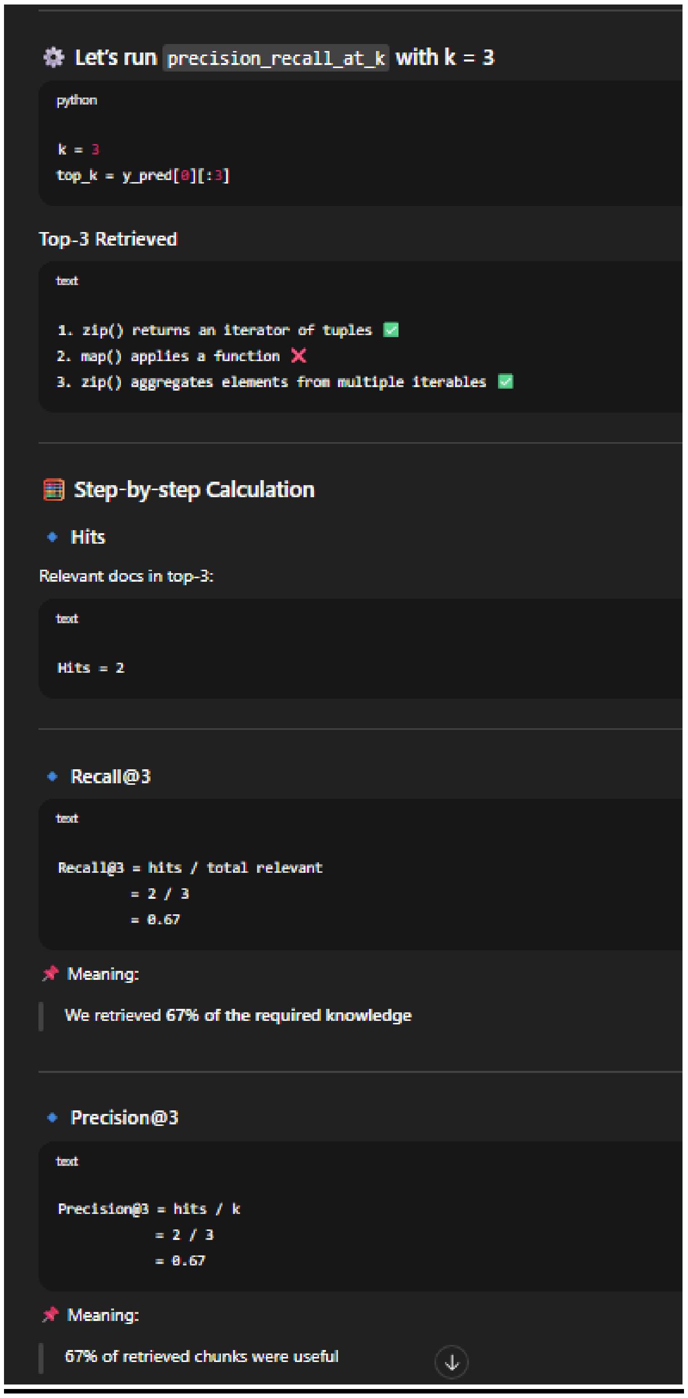
```python
y_true = [[
    "The zip() function returns an iterator of tuples",
    "zip() aggregates elements from multiple iterables",
    "zip() stops when the shortest iterable is exhausted"
]]
```

👉 There are **3 relevant chunks.**

🔍 Retrieved Chunks (`y_pred`)

Your vector database returns this ranked list:

```python
y_pred = [[
    "The zip() function returns an iterator of tuples",
    "map() applies a function to all items",
    "zip() aggregates elements from multiple iterables",
    "list comprehension syntax in Python",
    "zip() stops when the shortest iterable is exhausted"
]]
```



⚙️ Let's run `precision_recall_at_k` with k = 3

python
```python
k = 3
top_k = y_pred[0][:3]
```

Top-3 Retrieved

text
```text
1. zip() returns an iterator of tuples ✅
2. map() applies a function ❌
3. zip() aggregates elements from multiple iterables ✅
```

🧮 Step-by-step Calculation

🔹 Hits

Relevant docs in top-3:

text
```text
Hits = 2
```

🔹 Recall@3

text
```text
Recall@3 = hits / total relevant
         = 2 / 3
         = 0.67
```

📌 Meaning:

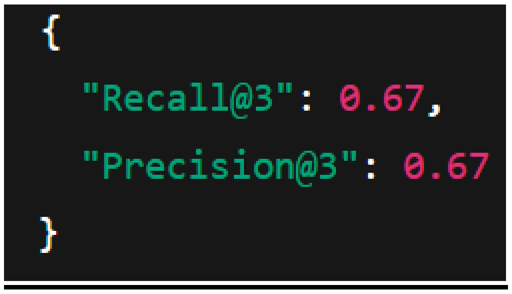
> We retrieved 67% of the required knowledge

🔹 Precision@3

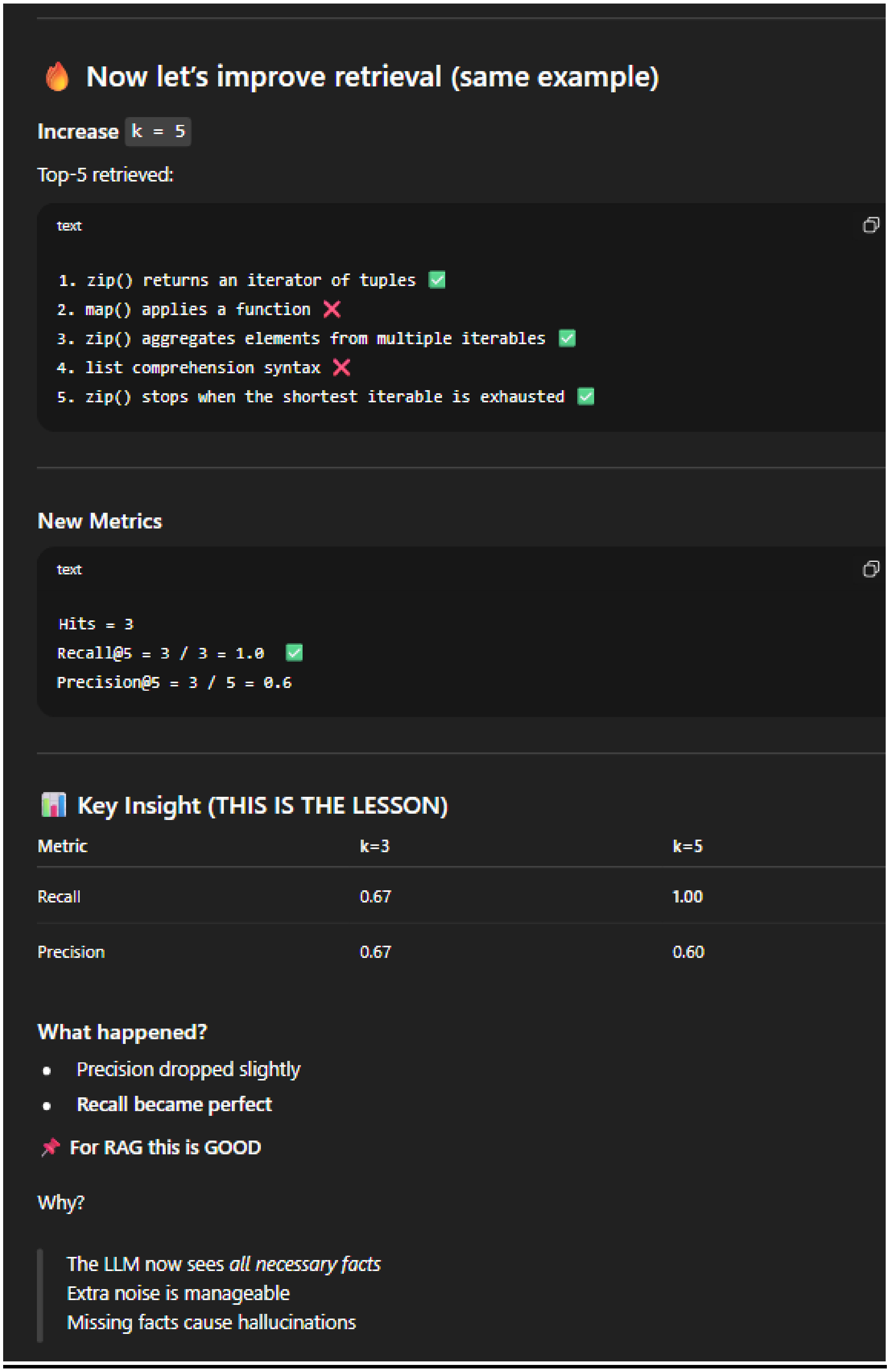
text
```text
Precision@3 = hits / k
            = 2 / 3
            = 0.67
```

📌 Meaning:

> 67% of retrieved chunks were useful



```
{
  "Recall@3": 0.67,
  "Precision@3": 0.67
}
```



🔥 Now let's improve retrieval (same example)

Increase `k = 5`

Top-5 retrieved:

```text
1. zip() returns an iterator of tuples ✅
2. map() applies a function ❌
3. zip() aggregates elements from multiple iterables ✅
4. list comprehension syntax ❌
5. zip() stops when the shortest iterable is exhausted ✅
```

New Metrics

```text
Hits = 3
Recall@5 = 3 / 3 = 1.0  ✅
Precision@5 = 3 / 5 = 0.6
```

📊 Key Insight (THIS IS THE LESSON)

Metric	k=3	k=5
Recall	0.67	**1.00**
Precision	0.67	0.60

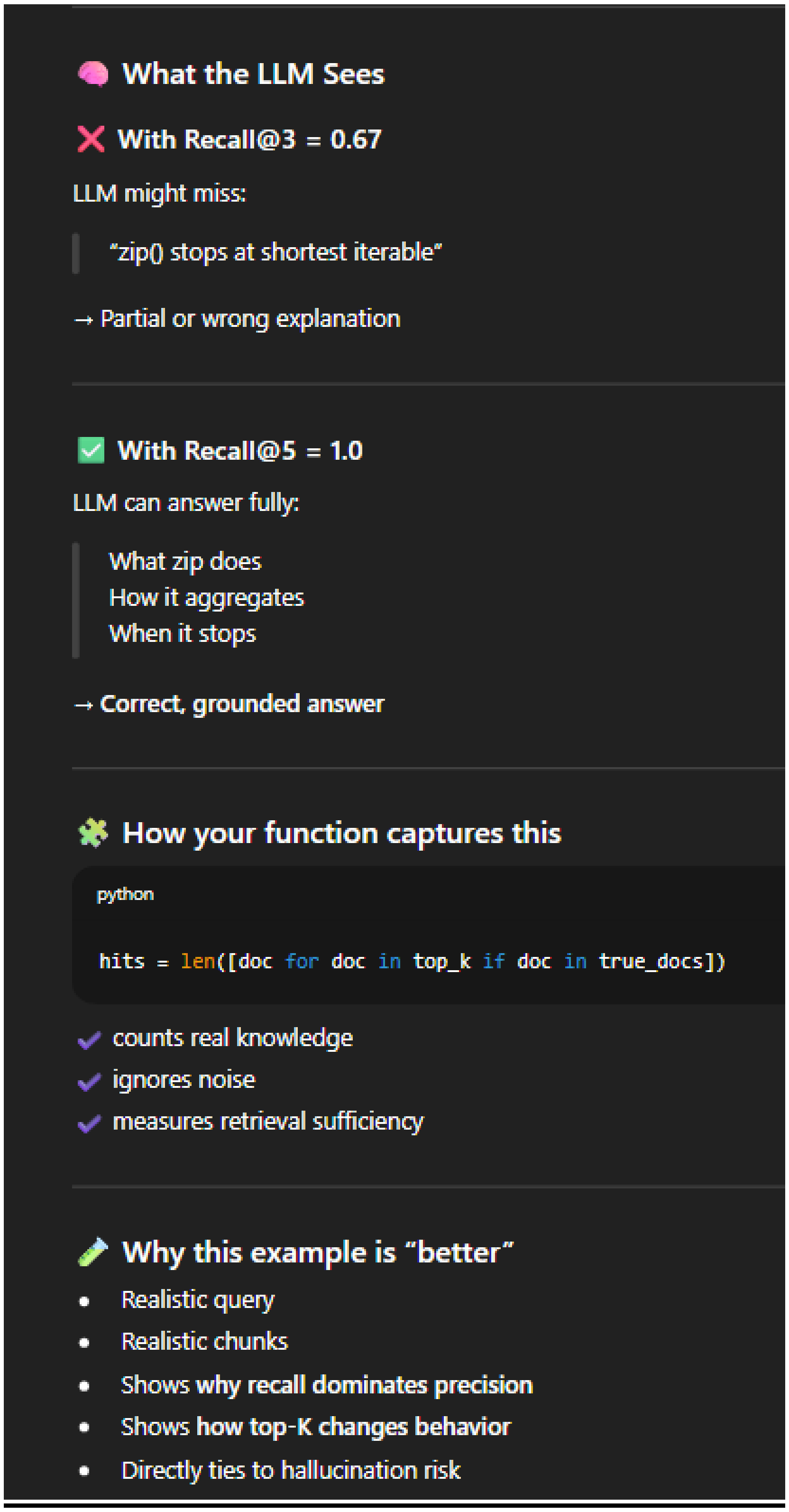
What happened?

- Precision dropped slightly
- **Recall became perfect**

📌 **For RAG this is GOOD**

Why?

> The LLM now sees *all necessary facts*
> Extra noise is manageable
> Missing facts cause hallucinations



🧠 What the LLM Sees

❌ With Recall@3 = 0.67

LLM might miss:

> "zip() stops at shortest iterable"

→ Partial or wrong explanation

✅ With Recall@5 = 1.0

LLM can answer fully:

> What zip does
> How it aggregates
> When it stops

→ **Correct, grounded answer**

🧩 How your function captures this

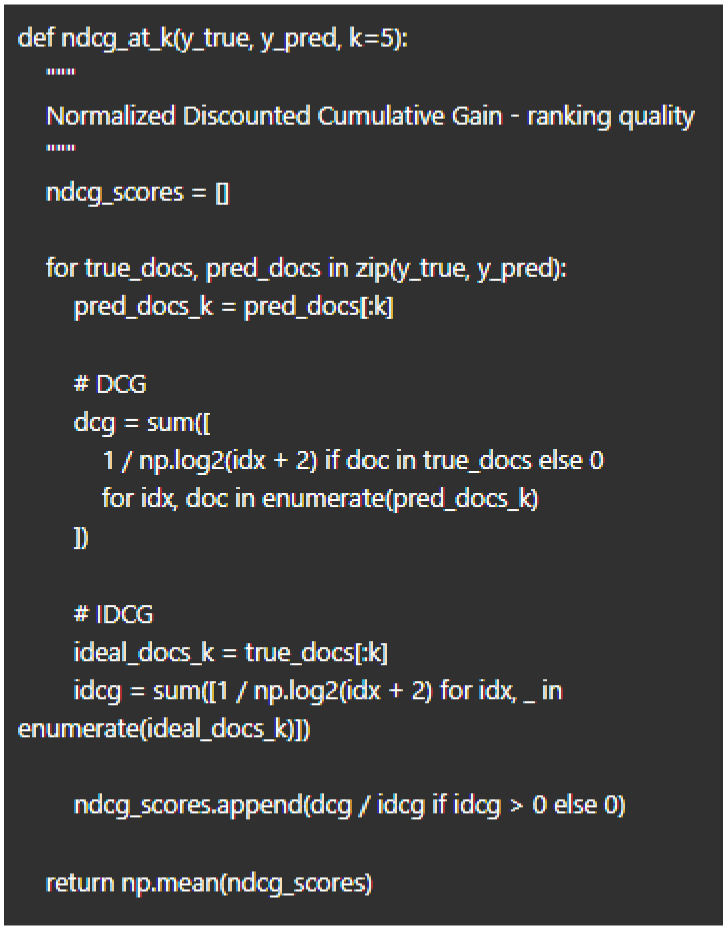
```python
hits = len([doc for doc in top_k if doc in true_docs])
```

✔ counts real knowledge
✔ ignores noise
✔ measures retrieval sufficiency

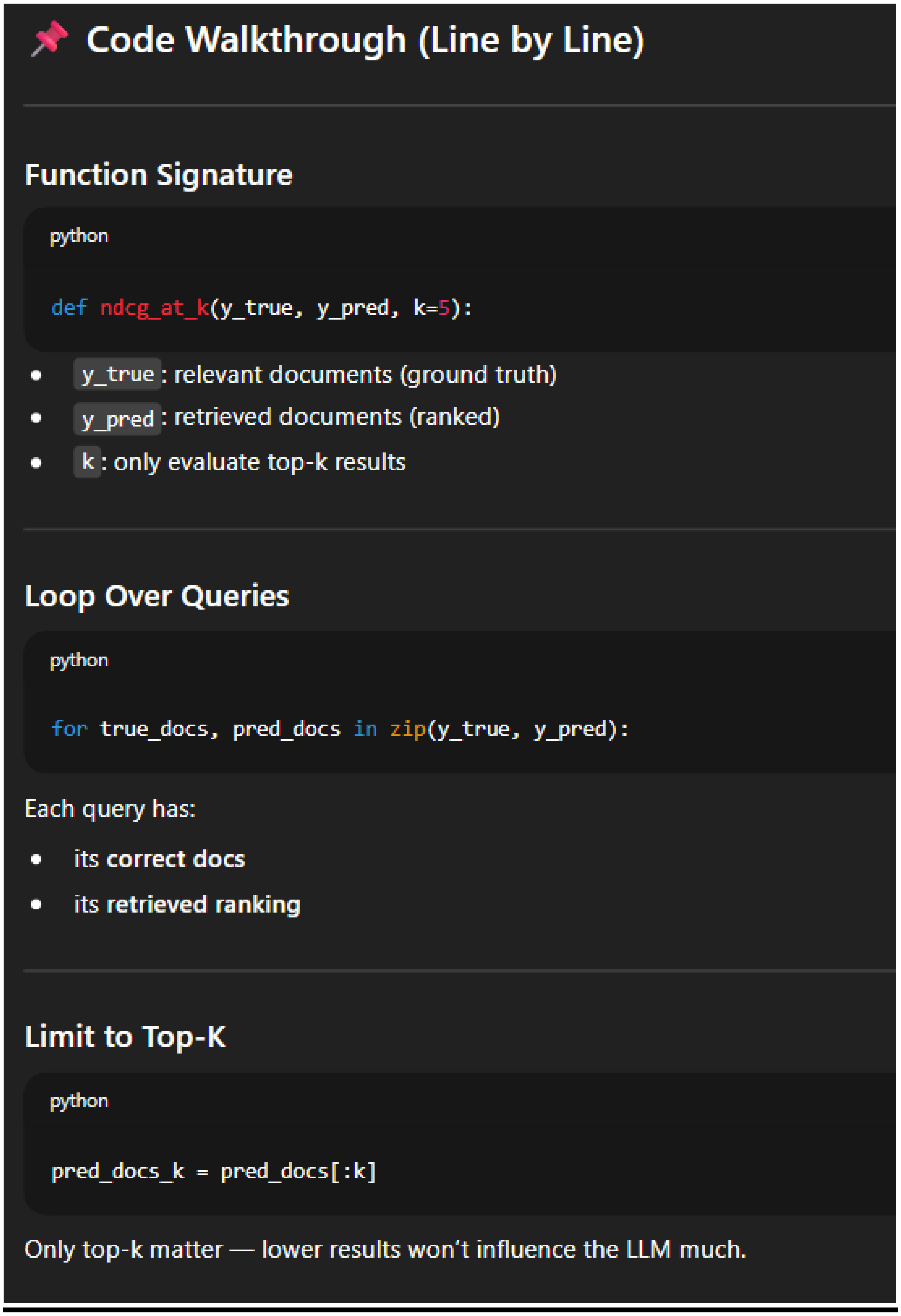
🧪 Why this example is "better"

- Realistic query
- Realistic chunks
- Shows **why recall dominates precision**
- Shows **how top-K changes behavior**
- Directly ties to hallucination risk

nDCG



```
def ndcg_at_k(y_true, y_pred, k=5):
    """
    Normalized Discounted Cumulative Gain - ranking quality
    """
    ndcg_scores = []

    for true_docs, pred_docs in zip(y_true, y_pred):
        pred_docs_k = pred_docs[:k]

        # DCG
        dcg = sum([
            1 / np.log2(idx + 2) if doc in true_docs else 0
            for idx, doc in enumerate(pred_docs_k)
        ])

        # IDCG
        ideal_docs_k = true_docs[:k]
        idcg = sum([1 / np.log2(idx + 2) for idx, _ in
enumerate(ideal_docs_k)])

        ndcg_scores.append(dcg / idcg if idcg > 0 else 0)

    return np.mean(ndcg_scores)
```



📌 Code Walkthrough (Line by Line)

Function Signature

python

```python
def ndcg_at_k(y_true, y_pred, k=5):
```

- `y_true` : relevant documents (ground truth)
- `y_pred` : retrieved documents (ranked)
- `k` : only evaluate top-k results

Loop Over Queries

python

```python
for true_docs, pred_docs in zip(y_true, y_pred):
```

Each query has:

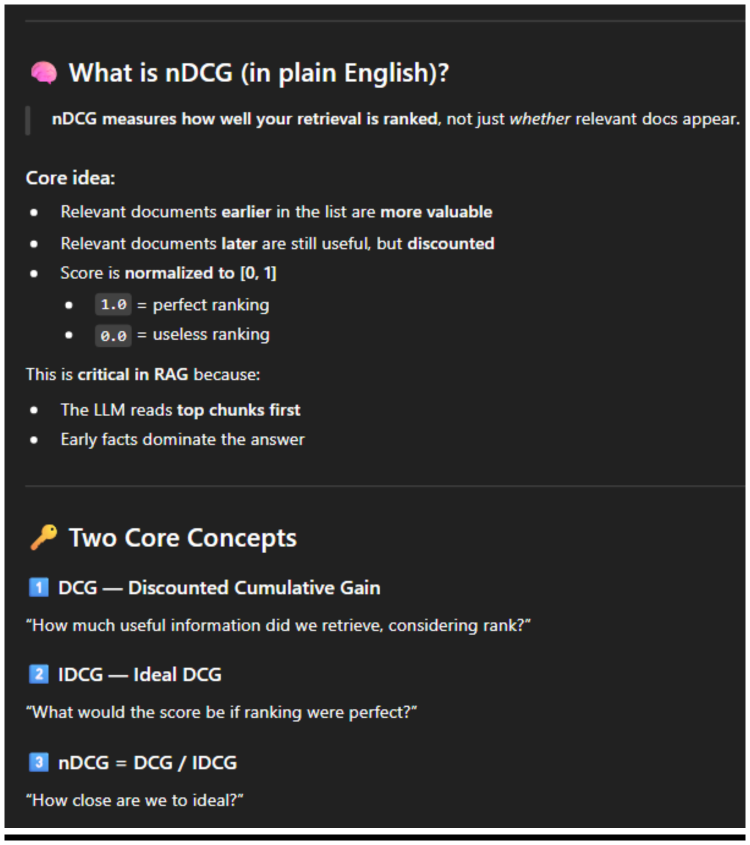
- its **correct docs**
- its **retrieved ranking**

Limit to Top-K

python

```python
pred_docs_k = pred_docs[:k]
```

Only top-k matter — lower results won't influence the LLM much.



🧠 What is nDCG (in plain English)?

> **nDCG measures how well your retrieval is ranked**, not just *whether* relevant docs appear.

Core idea:

- Relevant documents **earlier** in the list are **more valuable**
- Relevant documents **later** are still useful, but **discounted**
- Score is **normalized to [0, 1]**
  - `1.0` = perfect ranking
  - `0.0` = useless ranking

This is **critical in RAG** because:

- The LLM reads **top chunks first**
- Early facts dominate the answer

🔑 Two Core Concepts

1️⃣ DCG — Discounted Cumulative Gain

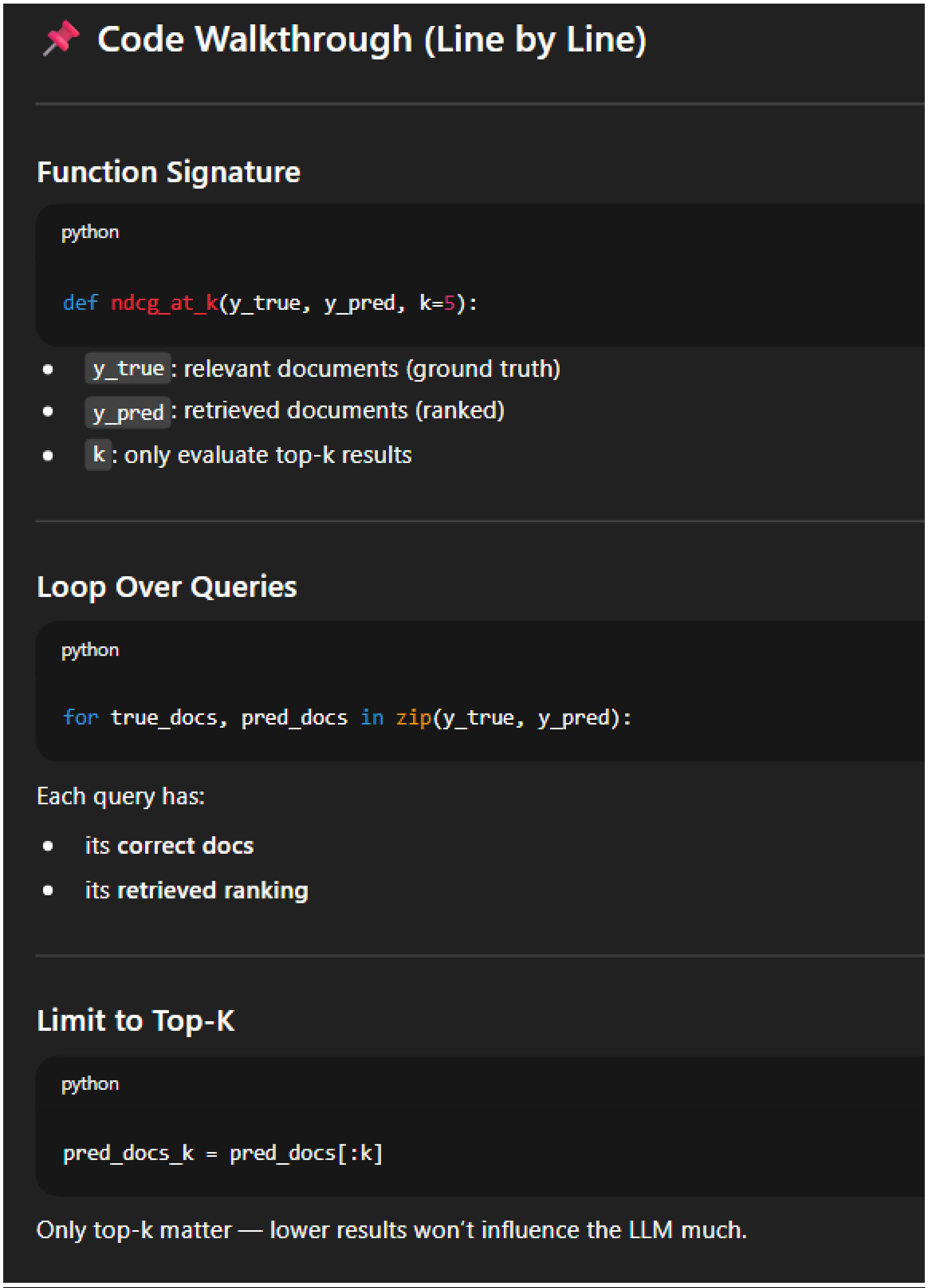
"How much useful information did we retrieve, considering rank?"

2️⃣ IDCG — Ideal DCG

"What would the score be if ranking were perfect?"

3️⃣ nDCG = DCG / IDCG

"How close are we to ideal?"



📌 Code Walkthrough (Line by Line)

Function Signature

```python
def ndcg_at_k(y_true, y_pred, k=5):
```

- `y_true` : relevant documents (ground truth)
- `y_pred` : retrieved documents (ranked)
- `k` : only evaluate top-k results

Loop Over Queries

```python
for true_docs, pred_docs in zip(y_true, y_pred):
```

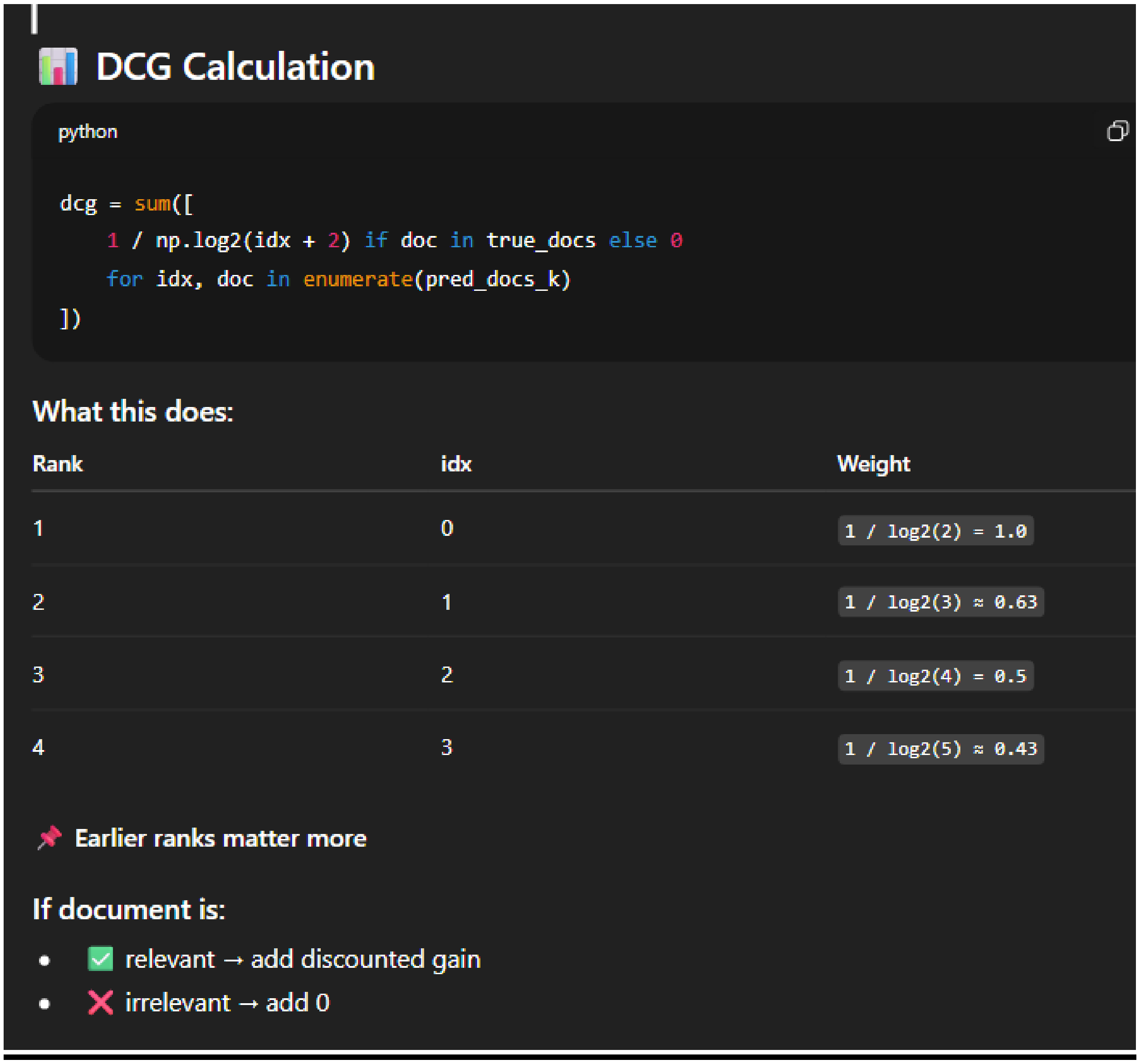
Each query has:

- its **correct docs**
- its **retrieved ranking**

Limit to Top-K

```python
pred_docs_k = pred_docs[:k]
```

Only top-k matter — lower results won't influence the LLM much.



📊 DCG Calculation

python

```python
dcg = sum([
    1 / np.log2(idx + 2) if doc in true_docs else 0
    for idx, doc in enumerate(pred_docs_k)
])
```

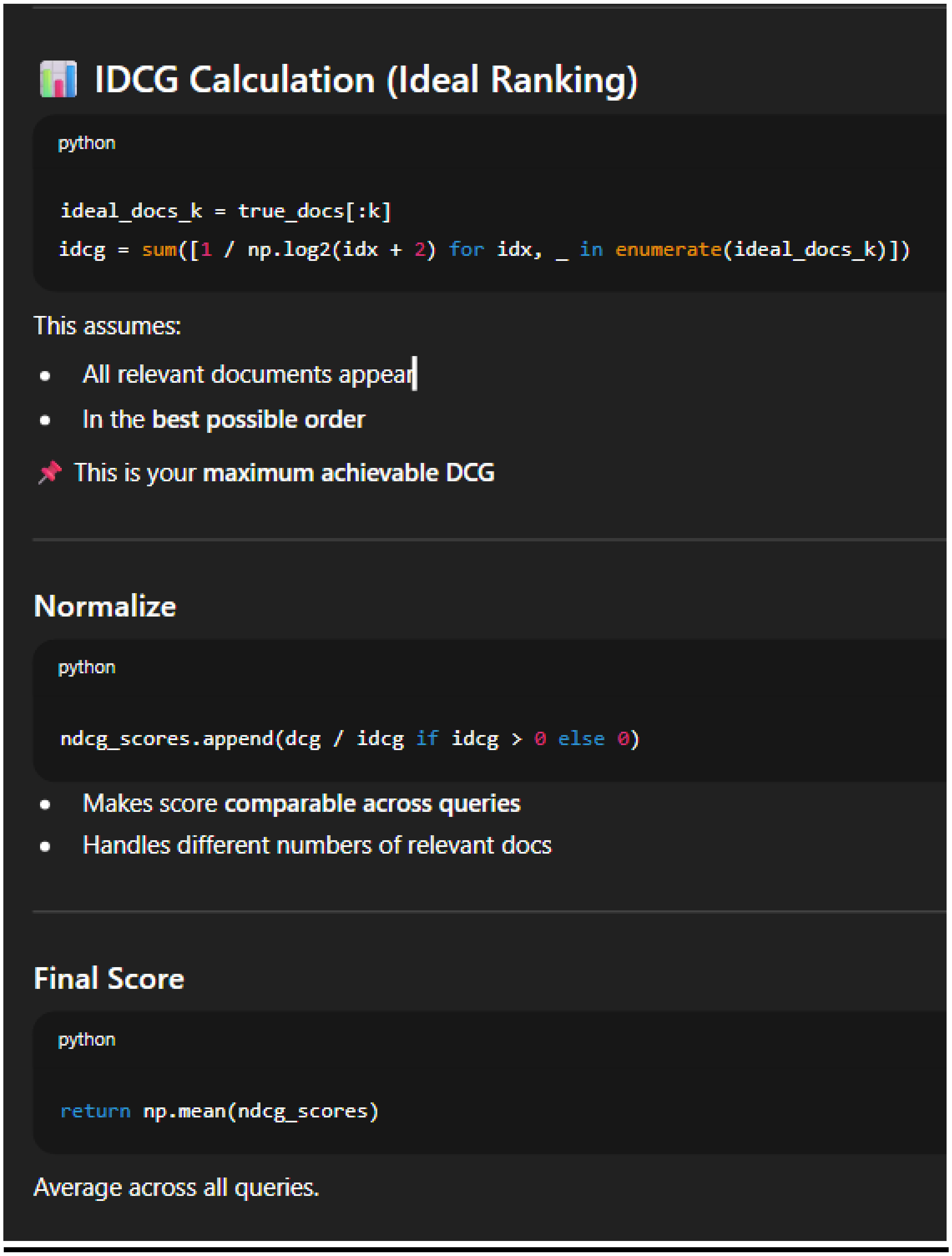
What this does:

Rank	idx	Weight
1	0	`1 / log2(2) = 1.0`
2	1	`1 / log2(3) ≈ 0.63`
3	2	`1 / log2(4) = 0.5`
4	3	`1 / log2(5) ≈ 0.43`

📌 **Earlier ranks matter more**

If document is:

- ✅ relevant → add discounted gain
- ❌ irrelevant → add 0



📊 IDCG Calculation (Ideal Ranking)

python

```python
ideal_docs_k = true_docs[:k]
idcg = sum([1 / np.log2(idx + 2) for idx, _ in enumerate(ideal_docs_k)])
```

This assumes:

- All relevant documents appear
- In the **best possible order**

📌 This is your **maximum achievable DCG**

Normalize

python

```python
ndcg_scores.append(dcg / idcg if idcg > 0 else 0)
```

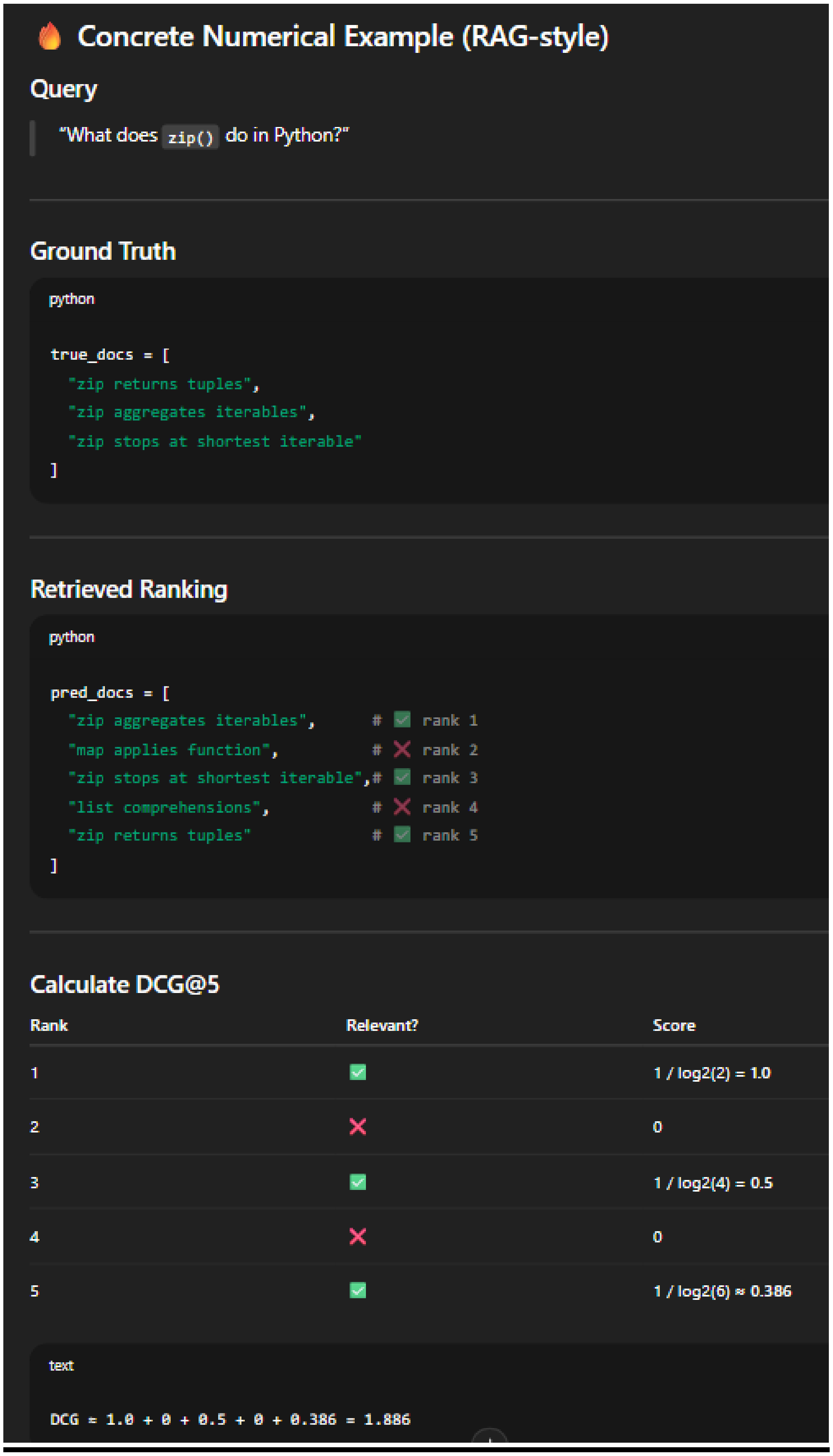
- Makes score **comparable across queries**
- Handles different numbers of relevant docs

Final Score

python

```python
return np.mean(ndcg_scores)
```

Average across all queries.



🔥 Concrete Numerical Example (RAG-style)

Query

> "What does `zip()` do in Python?"

Ground Truth

```python
true_docs = [
  "zip returns tuples",
  "zip aggregates iterables",
  "zip stops at shortest iterable"
]
```

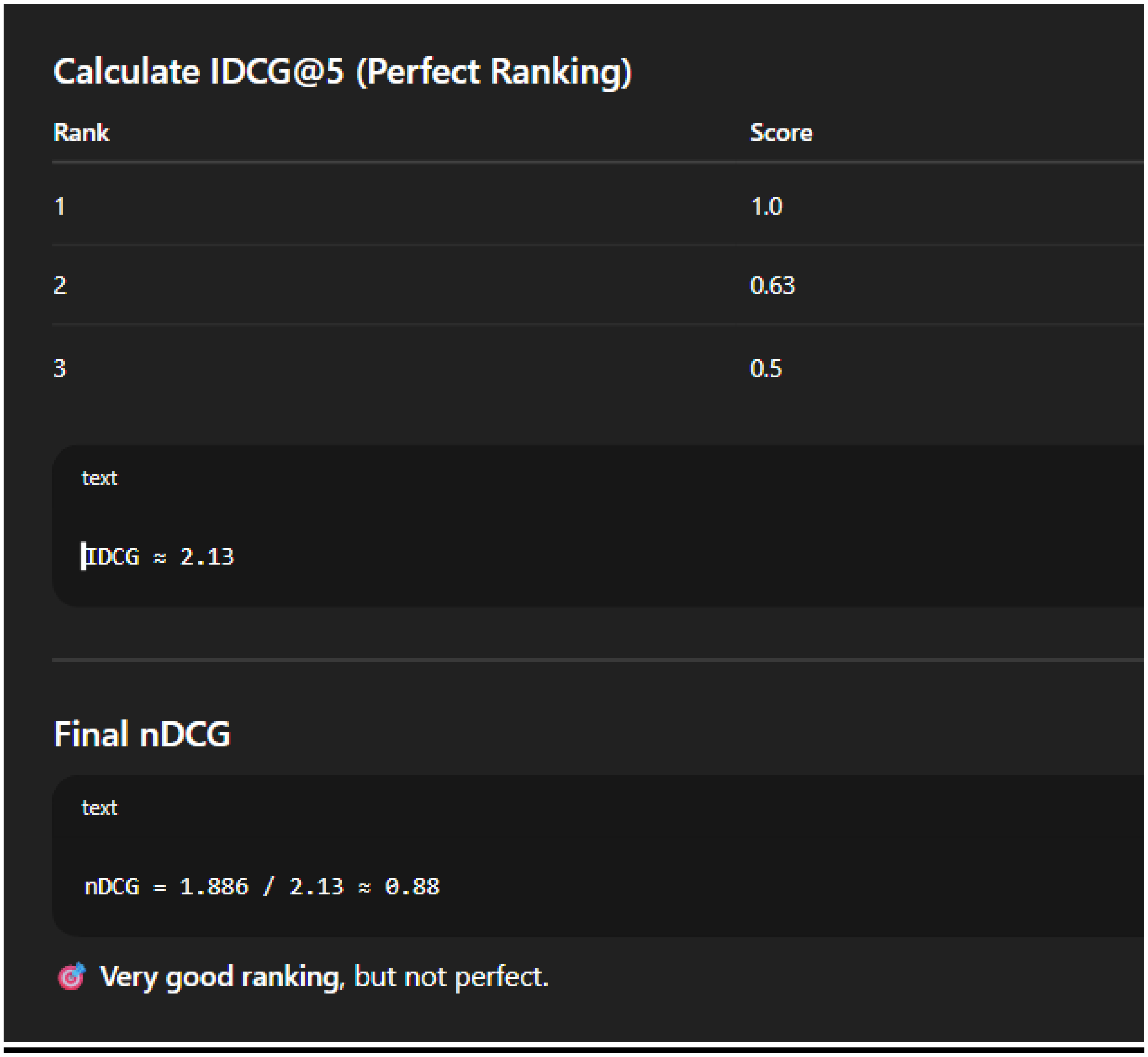
Retrieved Ranking

```python
pred_docs = [
  "zip aggregates iterables",      # ✅ rank 1
  "map applies function",          # ❌ rank 2
  "zip stops at shortest iterable",# ✅ rank 3
  "list comprehensions",           # ❌ rank 4
  "zip returns tuples"             # ✅ rank 5
]
```

Calculate DCG@5

Rank	Relevant?	Score
1	✅	1 / log2(2) = 1.0
2	❌	0
3	✅	1 / log2(4) = 0.5
4	❌	0
5	✅	1 / log2(6) ≈ 0.386

```text
DCG = 1.0 + 0 + 0.5 + 0 + 0.386 = 1.886
```



Calculate IDCG@5 (Perfect Ranking)

Rank	Score
1	1.0
2	0.63
3	0.5

text

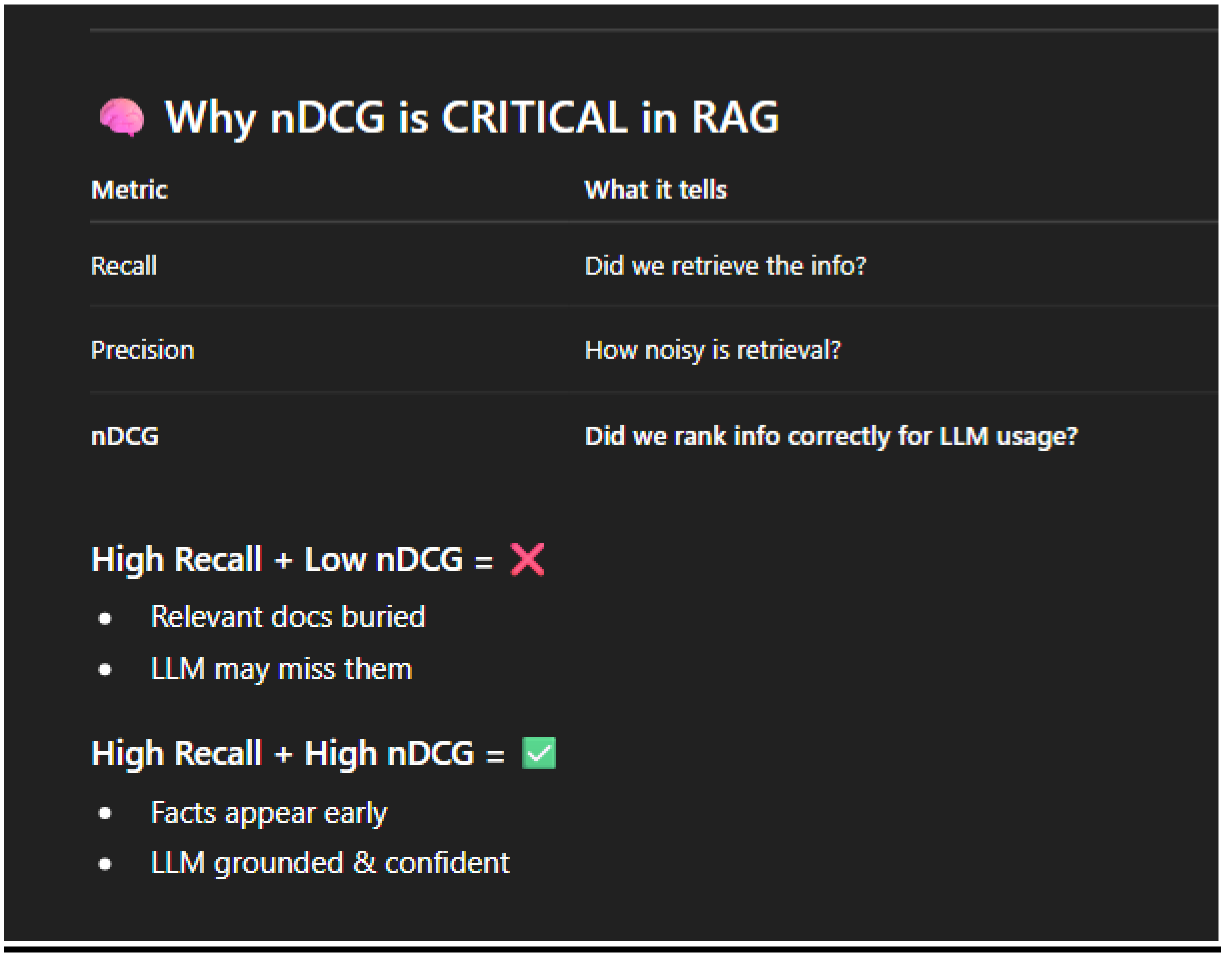
```
IDCG ≈ 2.13
```

Final nDCG

text

```
nDCG = 1.886 / 2.13 ≈ 0.88
```

🎯 **Very good ranking**, but not perfect.



🧠 Why nDCG is CRITICAL in RAG

Metric	What it tells
Recall	Did we retrieve the info?
Precision	How noisy is retrieval?
nDCG	**Did we rank info correctly for LLM usage?**

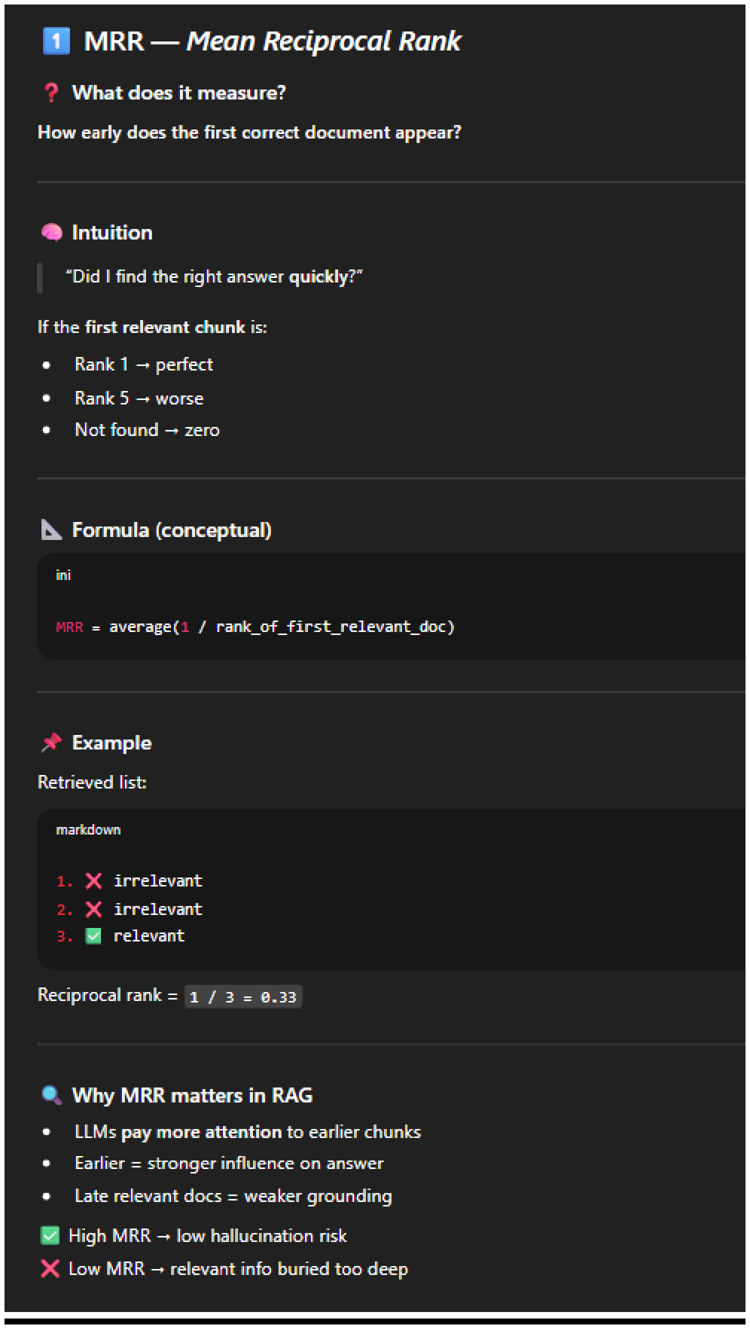
High Recall + Low nDCG = ❌

- Relevant docs buried
- LLM may miss them

High Recall + High nDCG = ✅

- Facts appear early
- LLM grounded & confident

Summary of metrics:



1️⃣ MRR — *Mean Reciprocal Rank*

❓ **What does it measure?**

How early does the first correct document appear?

🧠 **Intuition**

> "Did I find the right answer **quickly**?"

If the **first relevant chunk** is:

- Rank 1 → perfect
- Rank 5 → worse
- Not found → zero

📐 **Formula (conceptual)**

```ini
MRR = average(1 / rank_of_first_relevant_doc)
```

📌 **Example**

Retrieved list:

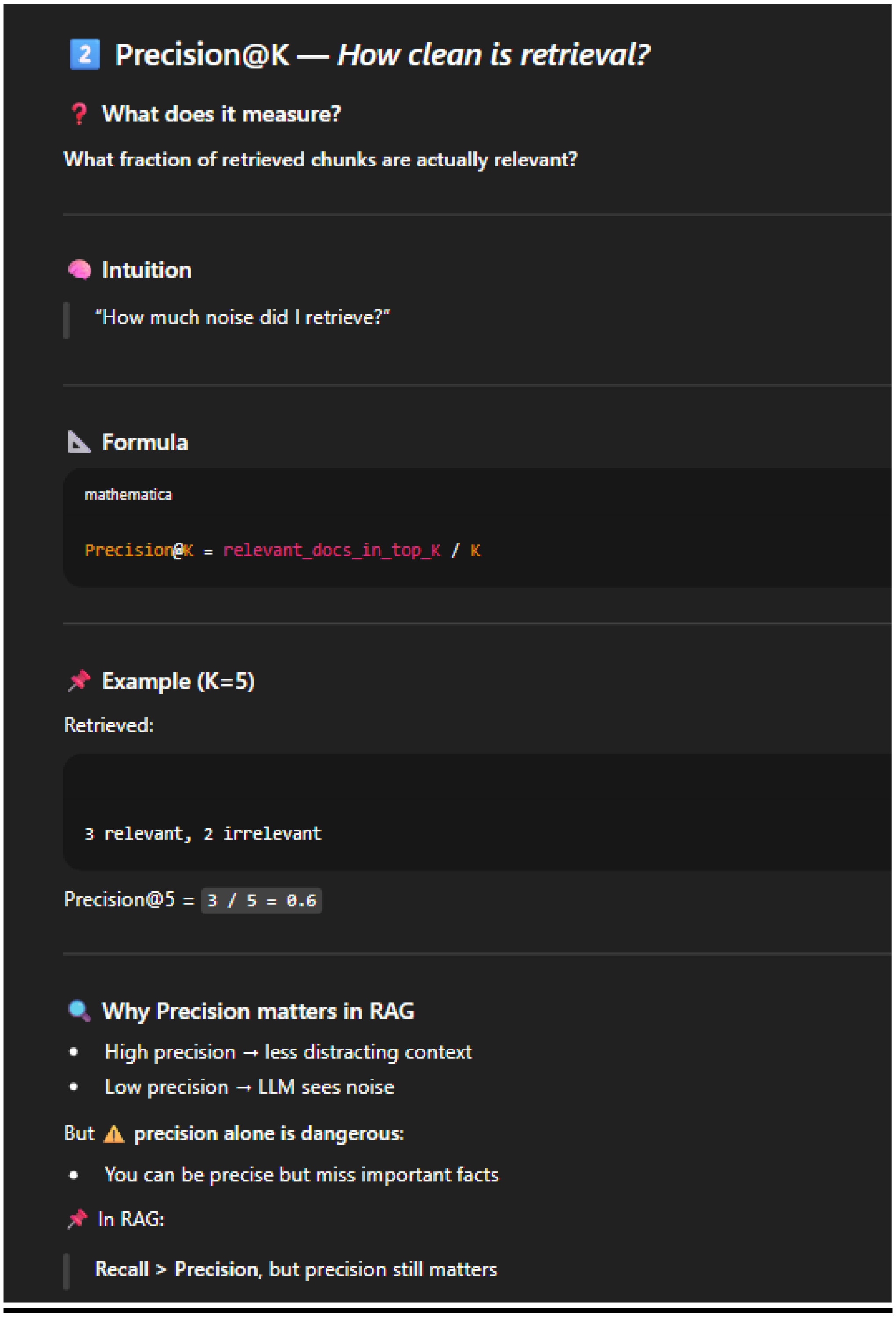
```markdown
1. ❌ irrelevant
2. ❌ irrelevant
3. ✅ relevant
```

Reciprocal rank = `1 / 3 = 0.33`

🔍 **Why MRR matters in RAG**

- LLMs **pay more attention** to earlier chunks
- Earlier = stronger influence on answer
- Late relevant docs = weaker grounding

✅ High MRR → low hallucination risk
❌ Low MRR → relevant info buried too deep



2️⃣ Precision@K — *How clean is retrieval?*

❓ What does it measure?

What fraction of retrieved chunks are actually relevant?

🧠 Intuition

> "How much noise did I retrieve?"

📐 Formula

```mathematica
Precision@K = relevant_docs_in_top_K / K
```

📌 Example (K=5)

Retrieved:

```
3 relevant, 2 irrelevant
```

Precision@5 = `3 / 5 = 0.6`

🔍 Why Precision matters in RAG

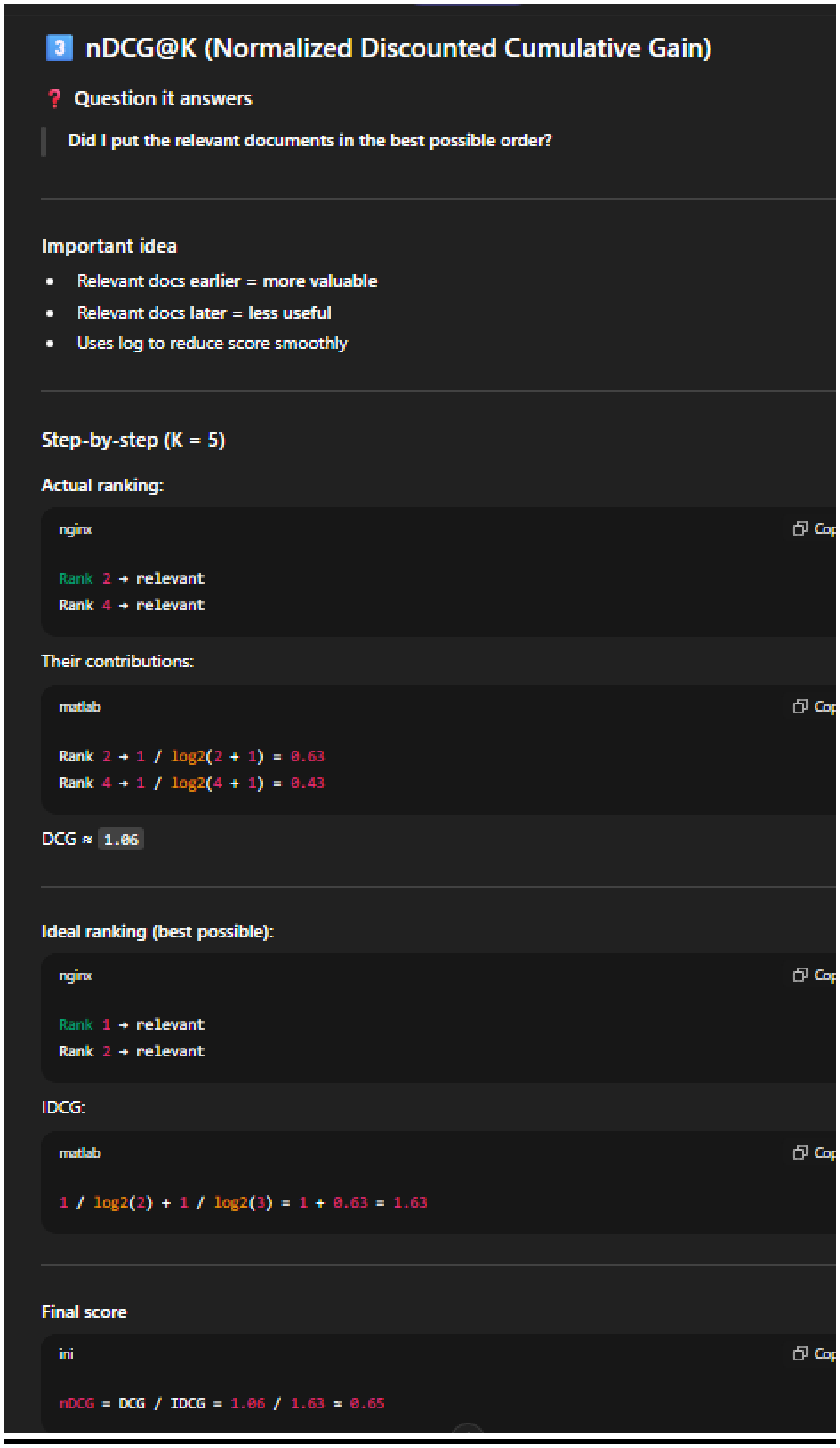
- High precision → less distracting context
- Low precision → LLM sees noise

But ⚠️ **precision alone is dangerous:**

- You can be precise but miss important facts

📌 In RAG:

> **Recall > Precision**, but precision still matters



3️⃣ nDCG@K (Normalized Discounted Cumulative Gain)

❓ Question it answers

> Did I put the relevant documents in the best possible order?

Important idea

- Relevant docs earlier = more valuable
- Relevant docs later = less useful
- Uses log to reduce score smoothly

Step-by-step (K = 5)

Actual ranking:

```nginx
Rank 2 → relevant
Rank 4 → relevant
```

Their contributions:

```matlab
Rank 2 → 1 / log2(2 + 1) = 0.63
Rank 4 → 1 / log2(4 + 1) = 0.43
```

DCG ≈ `1.06`

Ideal ranking (best possible):

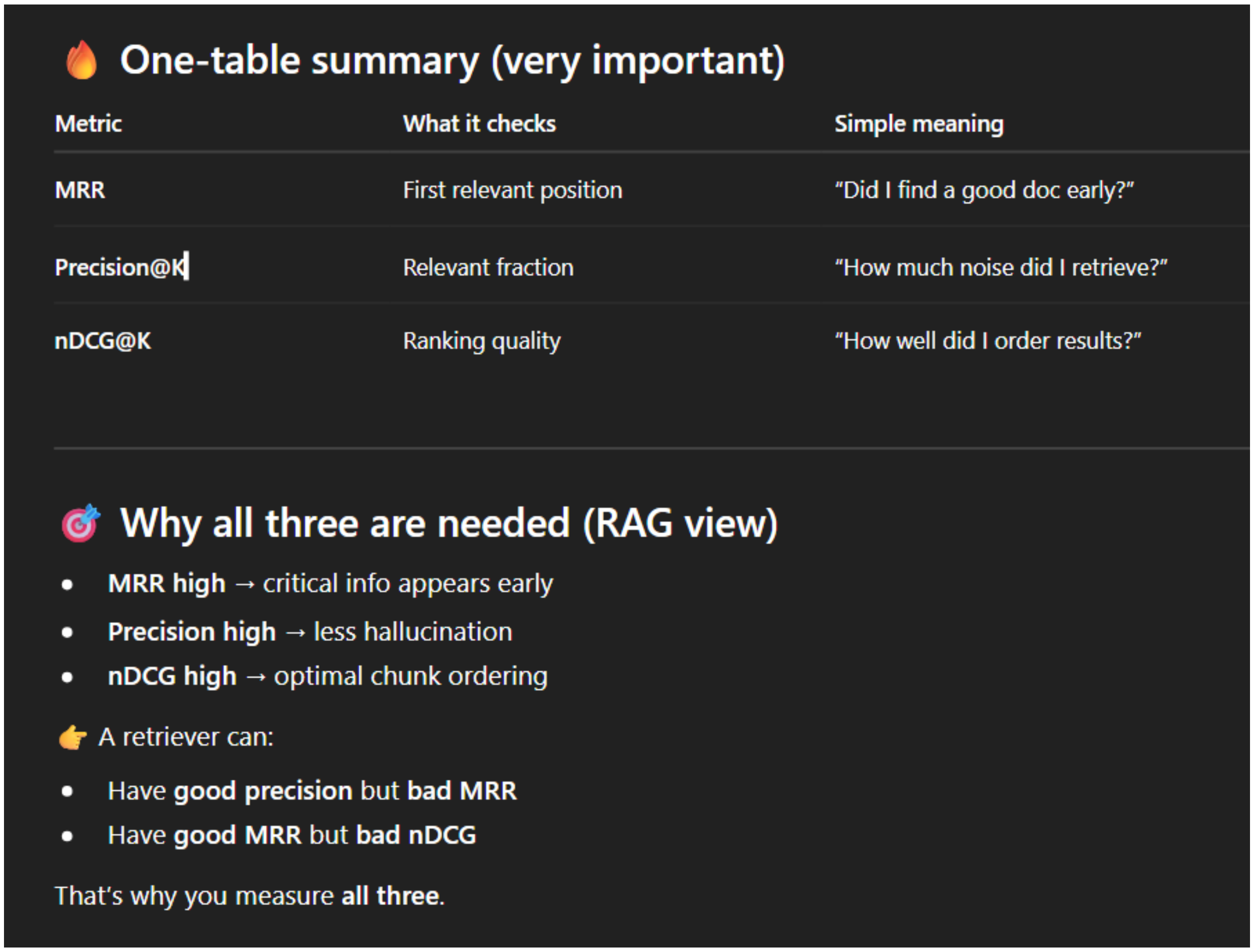
```nginx
Rank 1 → relevant
Rank 2 → relevant
```

IDCG:

```matlab
1 / log2(2) + 1 / log2(3) = 1 + 0.63 = 1.63
```

Final score

```ini
nDCG = DCG / IDCG = 1.06 / 1.63 = 0.65
```



🔥 One-table summary (very important)

Metric	What it checks	Simple meaning
MRR	First relevant position	"Did I find a good doc early?"
Precision@K	Relevant fraction	"How much noise did I retrieve?"
nDCG@K	Ranking quality	"How well did I order results?"

🎯 Why all three are needed (RAG view)

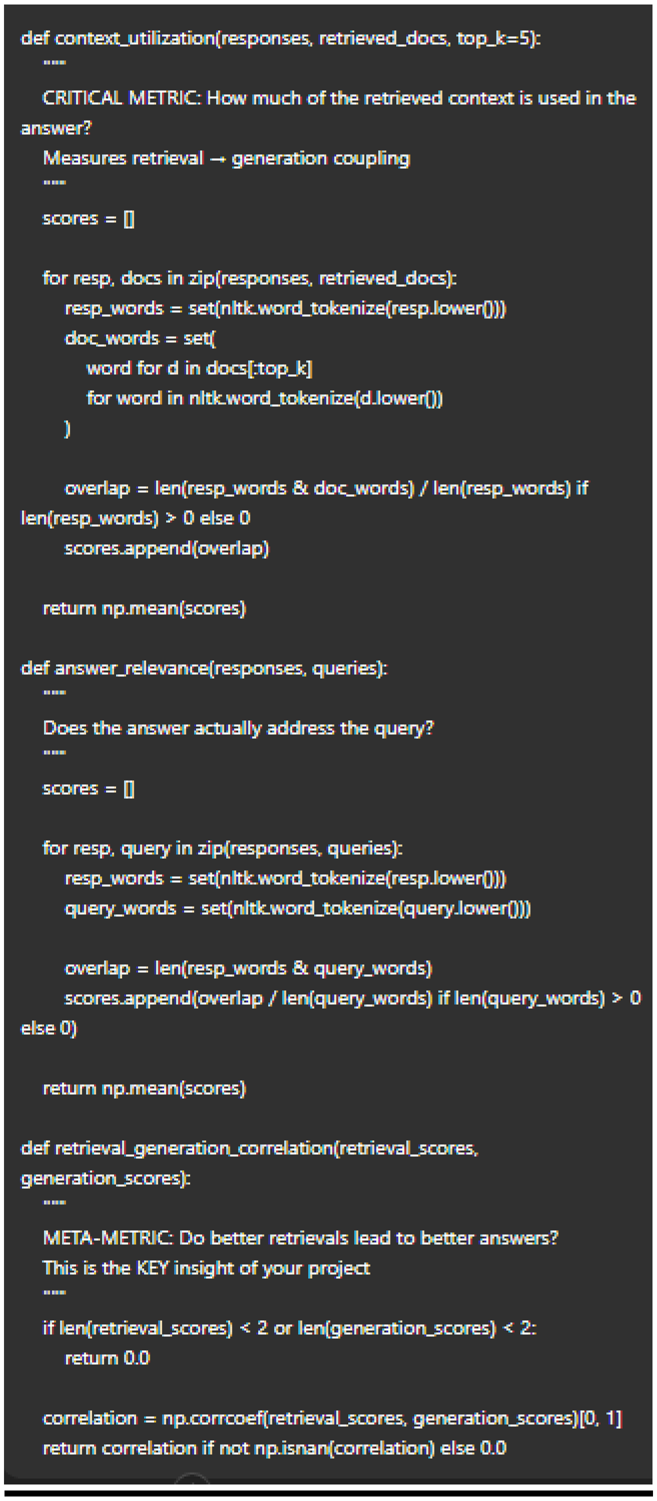
- **MRR high** → critical info appears early
- **Precision high** → less hallucination
- **nDCG high** → optimal chunk ordering

👉 A retriever can:

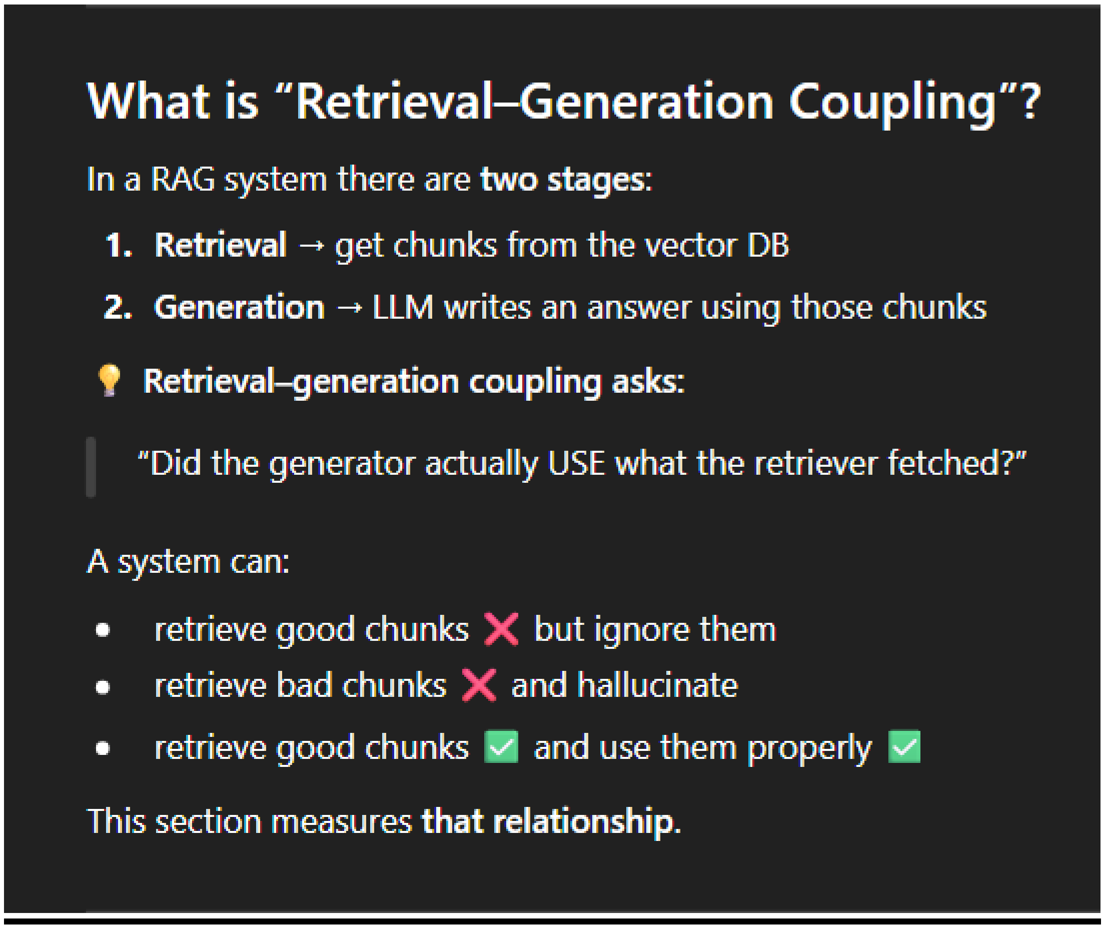
- Have **good precision** but **bad MRR**
- Have **good MRR** but **bad nDCG**

That's why you measure **all three**.

Retrieval–Generation Coupling



```
def context_utilization(responses, retrieved_docs, top_k=5):
    """
    CRITICAL METRIC: How much of the retrieved context is used in the answer?
    Measures retrieval → generation coupling
    """
    scores = []

    for resp, docs in zip(responses, retrieved_docs):
        resp_words = set(nltk.word_tokenize(resp.lower()))
        doc_words = set(
            word for d in docs[:top_k]
            for word in nltk.word_tokenize(d.lower())
        )

        overlap = len(resp_words & doc_words) / len(resp_words) if len(resp_words) > 0 else 0
        scores.append(overlap)

    return np.mean(scores)

def answer_relevance(responses, queries):
    """
    Does the answer actually address the query?
    """
    scores = []

    for resp, query in zip(responses, queries):
        resp_words = set(nltk.word_tokenize(resp.lower()))
        query_words = set(nltk.word_tokenize(query.lower()))

        overlap = len(resp_words & query_words)
        scores.append(overlap / len(query_words) if len(query_words) > 0 else 0)

    return np.mean(scores)

def retrieval_generation_correlation(retrieval_scores, generation_scores):
    """
    META-METRIC: Do better retrievals lead to better answers?
    This is the KEY insight of your project
    """
    if len(retrieval_scores) < 2 or len(generation_scores) < 2:
        return 0.0

    correlation = np.corrcoef(retrieval_scores, generation_scores)[0, 1]
    return correlation if not np.isnan(correlation) else 0.0
```



What is “Retrieval–Generation Coupling”?

In a RAG system there are **two stages**:

1. **Retrieval** → get chunks from the vector DB
2. **Generation** → LLM writes an answer using those chunks

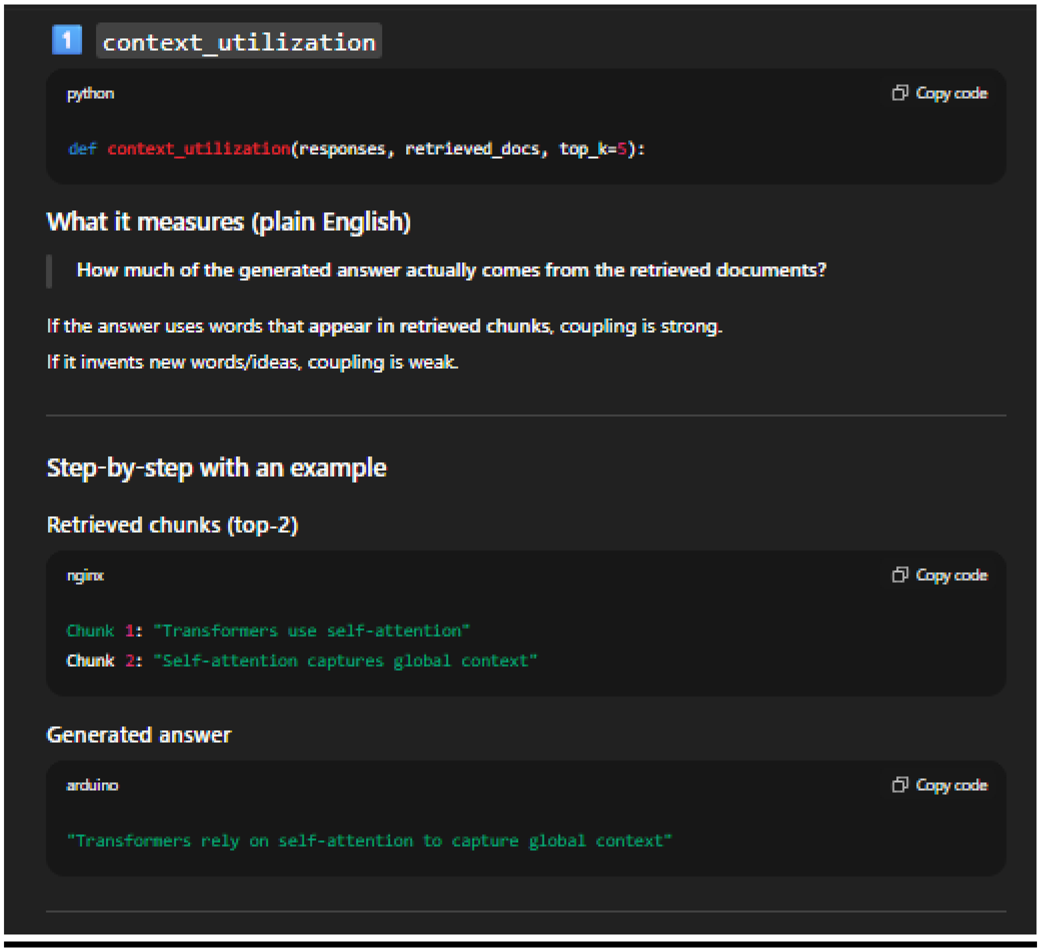
💡 **Retrieval–generation coupling asks:**

> “Did the generator actually USE what the retriever fetched?”

A system can:

- retrieve good chunks ❌ but ignore them
- retrieve bad chunks ❌ and hallucinate
- retrieve good chunks ✅ and use them properly ✅

This section measures **that relationship**.



1️⃣ `context_utilization`

```python
def context_utilization(responses, retrieved_docs, top_k=5):
```

What it measures (plain English)

> **How much of the generated answer actually comes from the retrieved documents?**

If the answer uses words that appear in retrieved chunks, coupling is strong.
If it invents new words/ideas, coupling is weak.

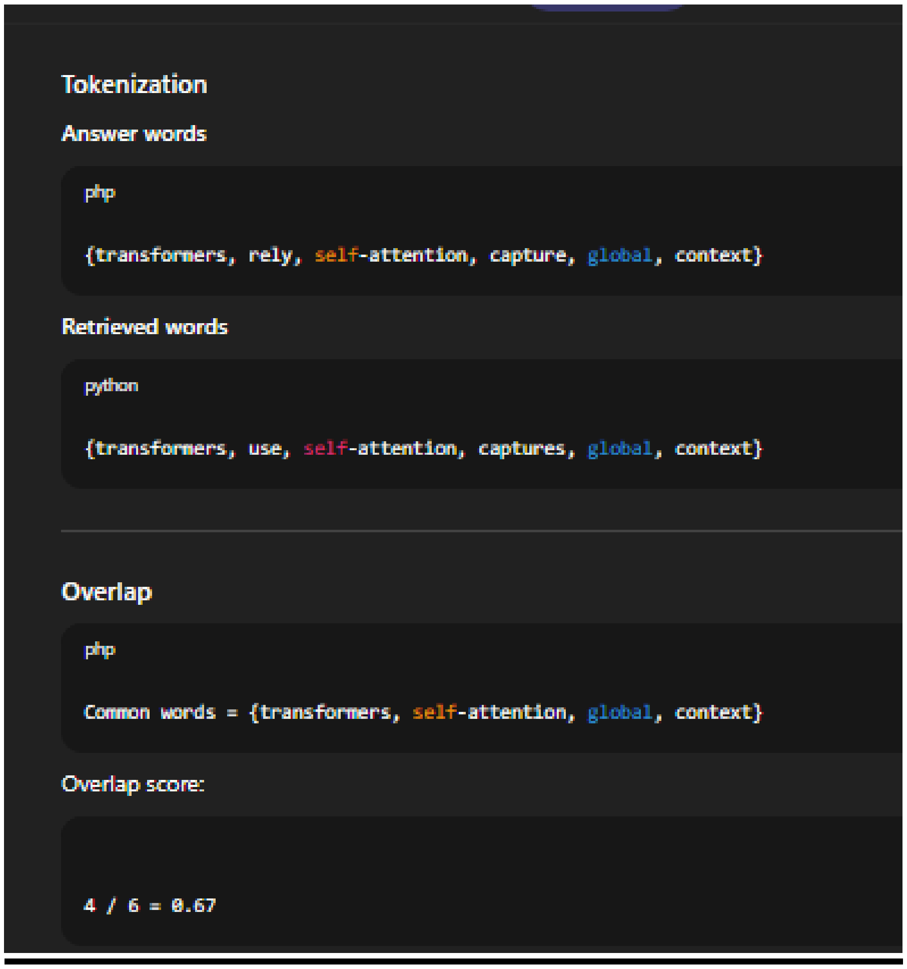
Step-by-step with an example

Retrieved chunks (top-2)

```nginx
Chunk 1: "Transformers use self-attention"
Chunk 2: "Self-attention captures global context"
```

Generated answer

```arduino
"Transformers rely on self-attention to capture global context"
```



Tokenization

Answer words

```php
{transformers, rely, self-attention, capture, global, context}
```

Retrieved words

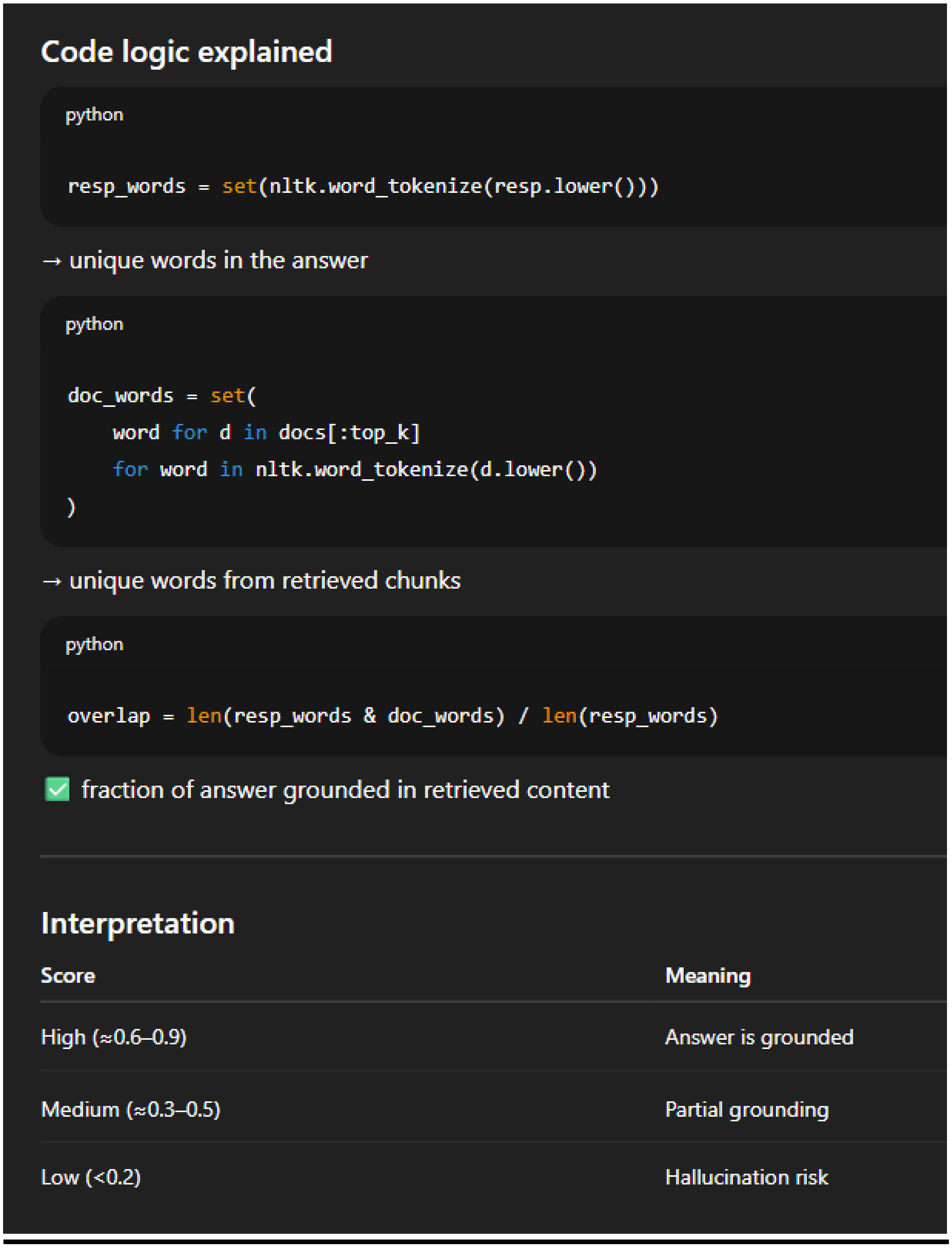
```python
{transformers, use, self-attention, captures, global, context}
```

Overlap

```php
Common words = {transformers, self-attention, global, context}
```

Overlap score:

```
4 / 6 = 0.67
```



Code logic explained

python

```python
resp_words = set(nltk.word_tokenize(resp.lower()))
```

→ unique words in the answer

python

```python
doc_words = set(
    word for d in docs[:top_k]
    for word in nltk.word_tokenize(d.lower())
)
```

→ unique words from retrieved chunks

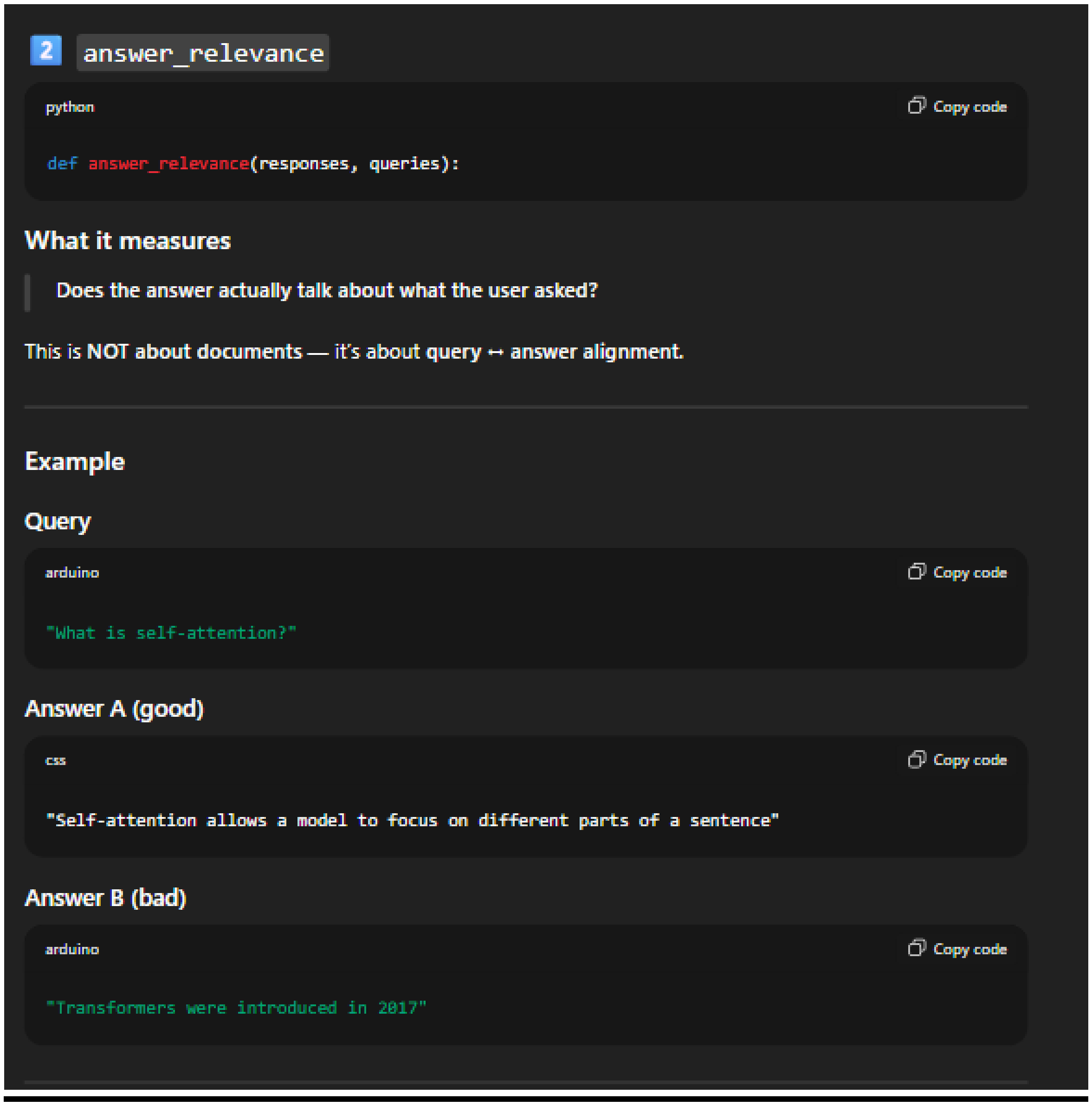
python

```python
overlap = len(resp_words & doc_words) / len(resp_words)
```

✅ fraction of answer grounded in retrieved content

Interpretation

Score	Meaning
High (≈0.6–0.9)	Answer is grounded
Medium (≈0.3–0.5)	Partial grounding
Low (<0.2)	Hallucination risk



2️⃣ `answer_relevance`

python

```python
def answer_relevance(responses, queries):
```

What it measures

> **Does the answer actually talk about what the user asked?**

This is **NOT about documents** — it's about **query ↔ answer alignment.**

Example

Query

arduino

```arduino
"What is self-attention?"
```

Answer A (good)

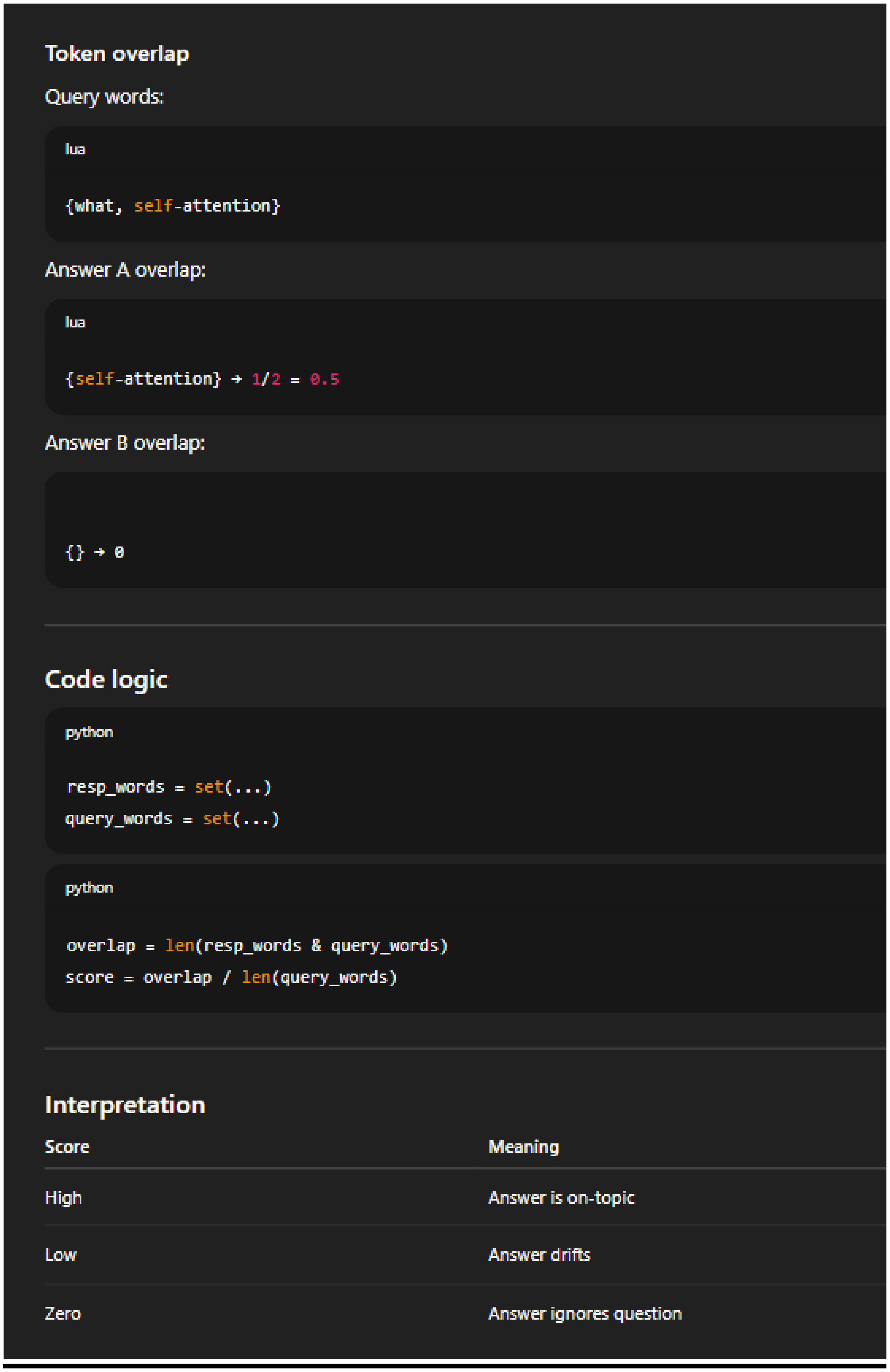
css

```css
"Self-attention allows a model to focus on different parts of a sentence"
```

Answer B (bad)

arduino

```arduino
"Transformers were introduced in 2017"
```



Token overlap

Query words:

```lua
{what, self-attention}
```

Answer A overlap:

```lua
{self-attention} → 1/2 = 0.5
```

Answer B overlap:

```
{} → 0
```

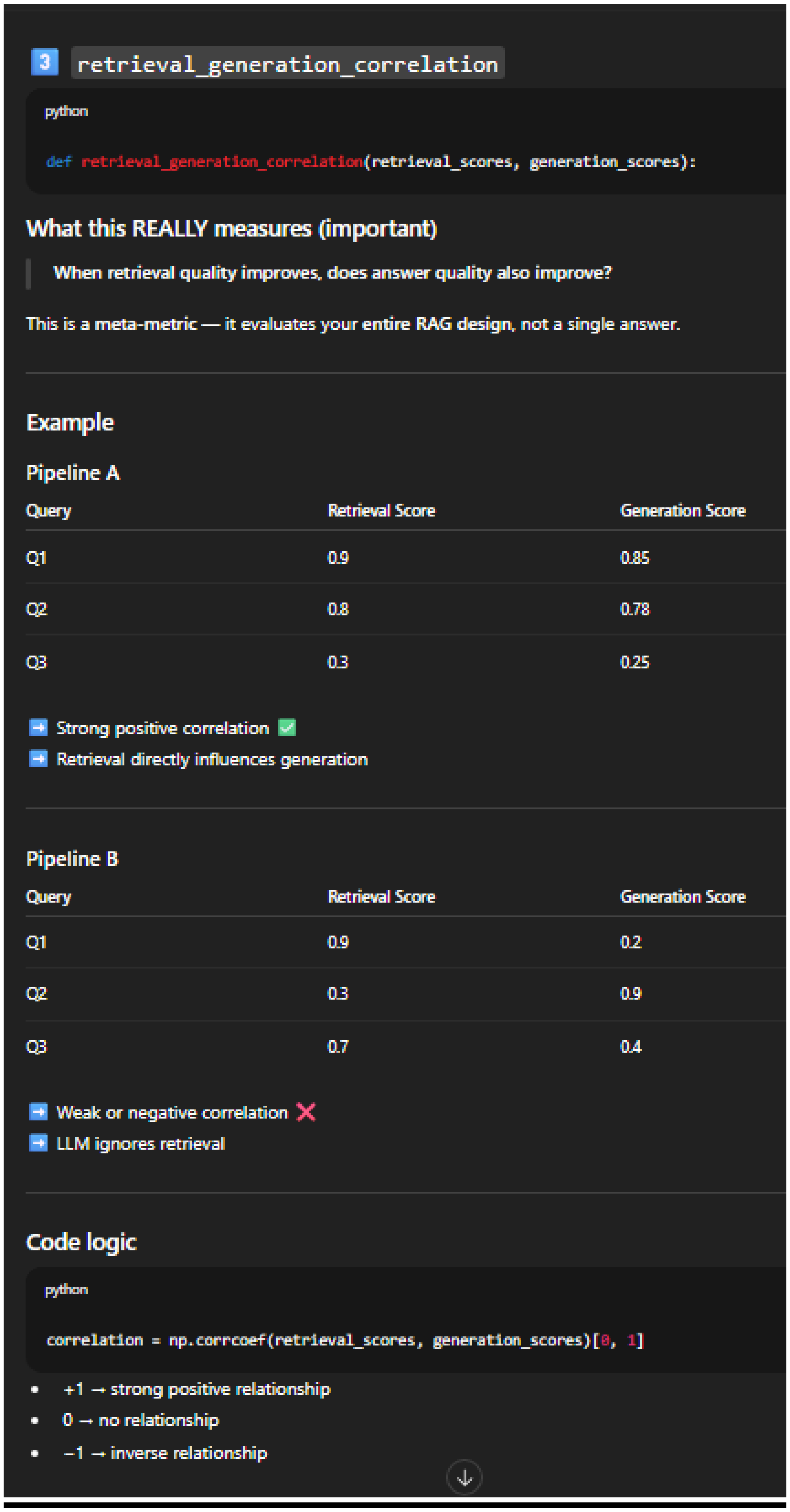
Code logic

```python
resp_words = set(...)
query_words = set(...)
```

```python
overlap = len(resp_words & query_words)
score = overlap / len(query_words)
```

Interpretation

Score	Meaning
High	Answer is on-topic
Low	Answer drifts
Zero	Answer ignores question



3️⃣ `retrieval_generation_correlation`

```python
def retrieval_generation_correlation(retrieval_scores, generation_scores):
```

What this REALLY measures (important)

> **When retrieval quality improves, does answer quality also improve?**

This is a **meta-metric** — it evaluates your **entire RAG design**, not a single answer.

Example

Pipeline A

Query	Retrieval Score	Generation Score
Q1	0.9	0.85
Q2	0.8	0.78
Q3	0.3	0.25

➡️ Strong positive correlation ✅
➡️ Retrieval directly influences generation

Pipeline B

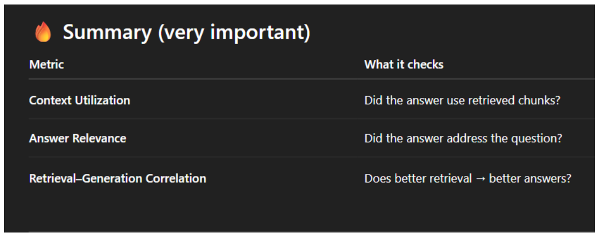
Query	Retrieval Score	Generation Score
Q1	0.9	0.2
Q2	0.3	0.9
Q3	0.7	0.4

➡️ Weak or negative correlation ❌
➡️ LLM ignores retrieval

Code logic

```python
correlation = np.corrcoef(retrieval_scores, generation_scores)[0, 1]
```

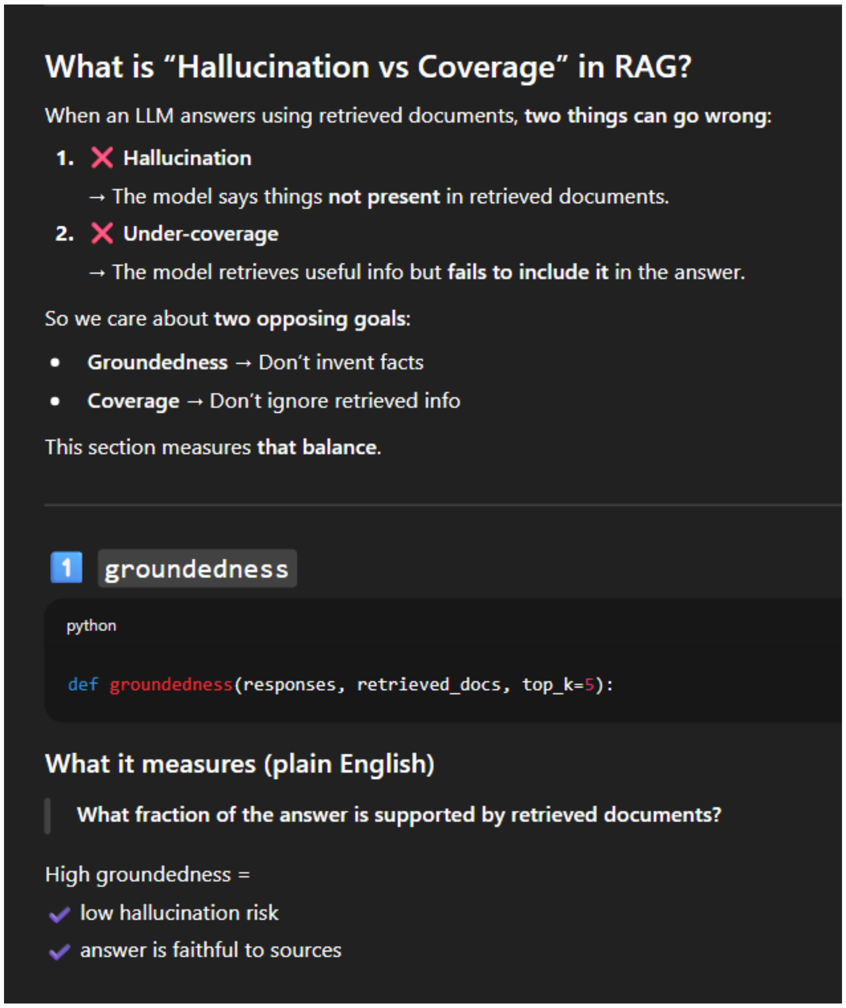
- +1 → strong positive relationship
- 0 → no relationship
- −1 → inverse relationship



🔥 Summary (very important)

Metric	What it checks
Context Utilization	Did the answer use retrieved chunks?
Answer Relevance	Did the answer address the question?
Retrieval–Generation Correlation	Does better retrieval → better answers?

Hallucination vs Coverage



What is “Hallucination vs Coverage” in RAG?

When an LLM answers using retrieved documents, **two things can go wrong**:

1. ❌ **Hallucination**
   → The model says things **not present** in retrieved documents.
2. ❌ **Under-coverage**
   → The model retrieves useful info but **fails to include it** in the answer.

So we care about **two opposing goals**:

- **Groundedness** → Don’t invent facts
- **Coverage** → Don’t ignore retrieved info

This section measures **that balance**.

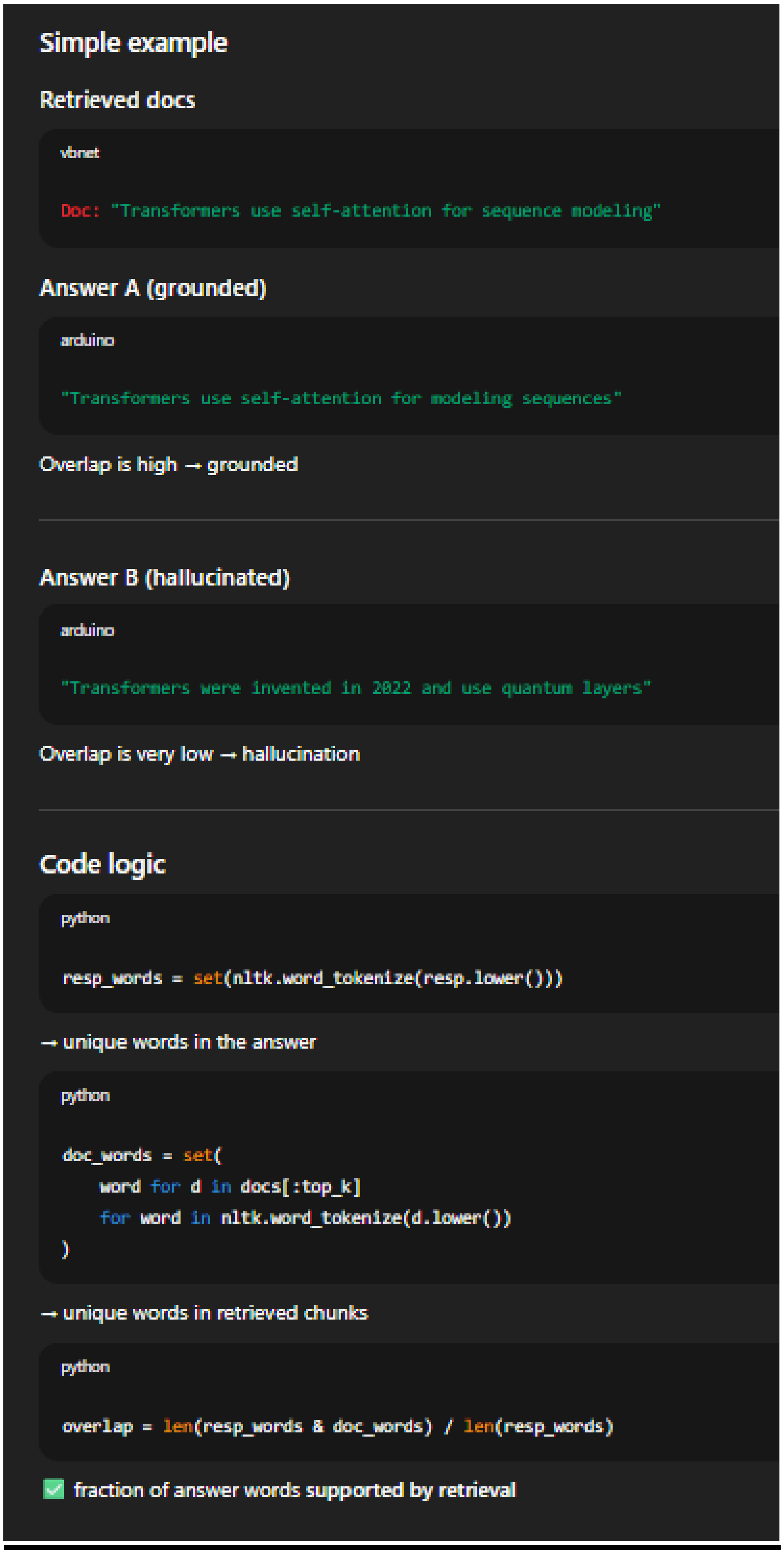
1️⃣ `groundedness`

```python
def groundedness(responses, retrieved_docs, top_k=5):
```

What it measures (plain English)

> **What fraction of the answer is supported by retrieved documents?**

High groundedness =
✔ low hallucination risk
✔ answer is faithful to sources



Simple example

Retrieved docs

vbnet

```vbnet
Doc: "Transformers use self-attention for sequence modeling"
```

Answer A (grounded)

arduino

```arduino
"Transformers use self-attention for modeling sequences"
```

Overlap is high → grounded

Answer B (hallucinated)

arduino

```arduino
"Transformers were invented in 2022 and use quantum layers"
```

Overlap is very low → hallucination

Code logic

python

```python
resp_words = set(nltk.word_tokenize(resp.lower()))
```

→ unique words in the answer

python

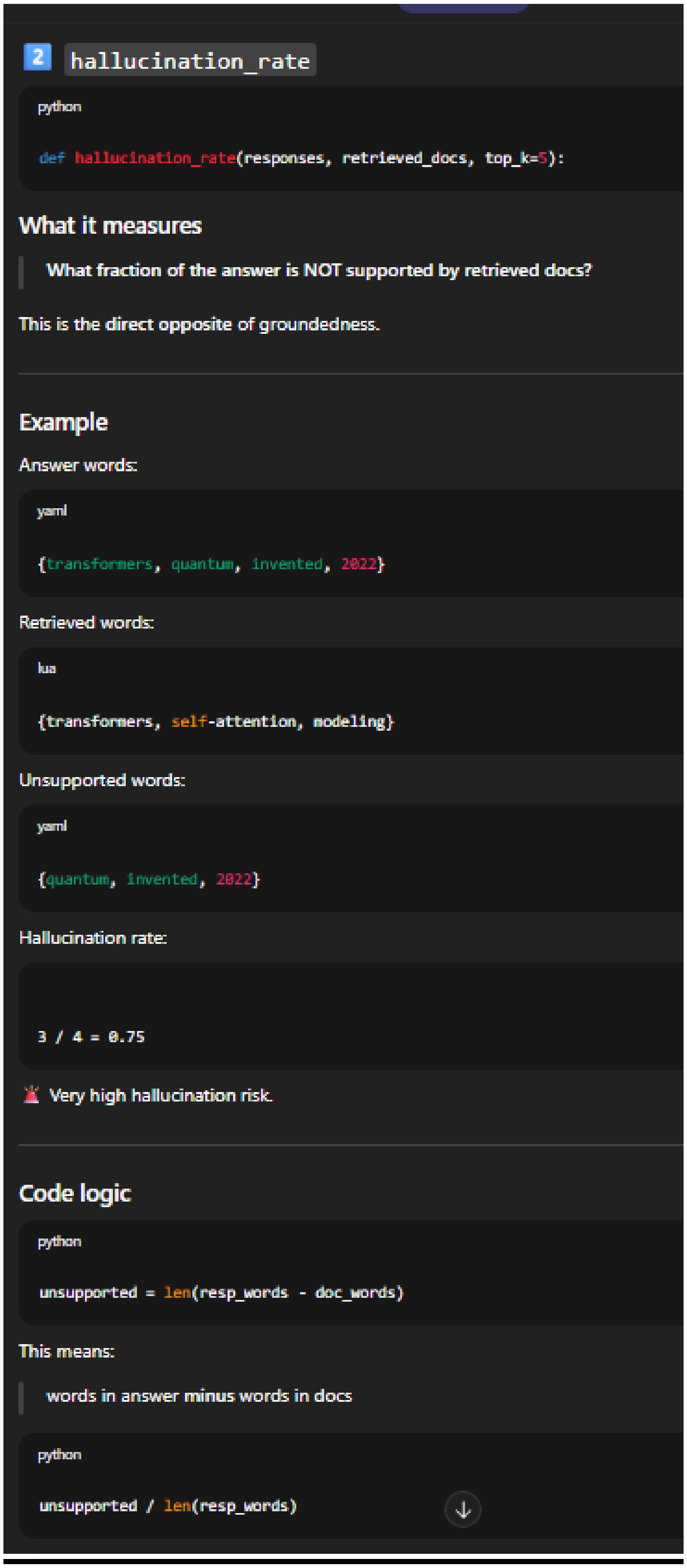
```python
doc_words = set(
    word for d in docs[:top_k]
    for word in nltk.word_tokenize(d.lower())
)
```

→ unique words in retrieved chunks

python

```python
overlap = len(resp_words & doc_words) / len(resp_words)
```

✅ fraction of answer words supported by retrieval



2️⃣ `hallucination_rate`

python

```python
def hallucination_rate(responses, retrieved_docs, top_k=5):
```

What it measures

> What fraction of the answer is NOT supported by retrieved docs?

This is the direct opposite of groundedness.

Example

Answer words:

yaml

```yaml
{transformers, quantum, invented, 2022}
```

Retrieved words:

lua

```lua
{transformers, self-attention, modeling}
```

Unsupported words:

yaml

```yaml
{quantum, invented, 2022}
```

Hallucination rate:

```
3 / 4 = 0.75
```

🚨 Very high hallucination risk.

Code logic

python

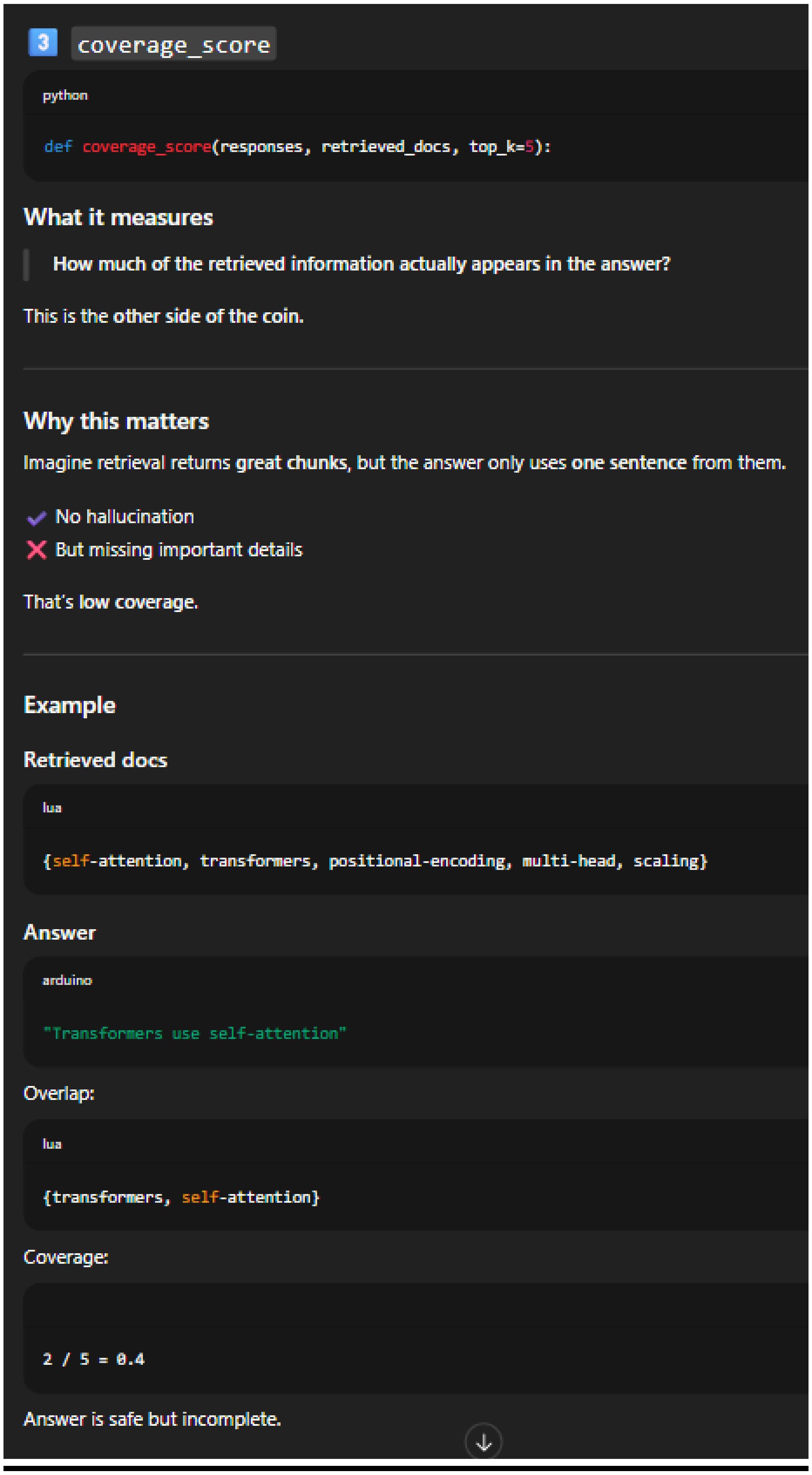
```python
unsupported = len(resp_words - doc_words)
```

This means:

> words in answer minus words in docs

python

```python
unsupported / len(resp_words)
```



3 `coverage_score`

python

```python
def coverage_score(responses, retrieved_docs, top_k=5):
```

What it measures

> **How much of the retrieved information actually appears in the answer?**

This is the **other side of the coin.**

Why this matters

Imagine retrieval returns **great chunks**, but the answer only uses **one sentence** from them.

✓ No hallucination
✗ But missing important details

That's **low coverage.**

Example

Retrieved docs

lua

```lua
{self-attention, transformers, positional-encoding, multi-head, scaling}
```

Answer

arduino

```arduino
"Transformers use self-attention"
```

Overlap:

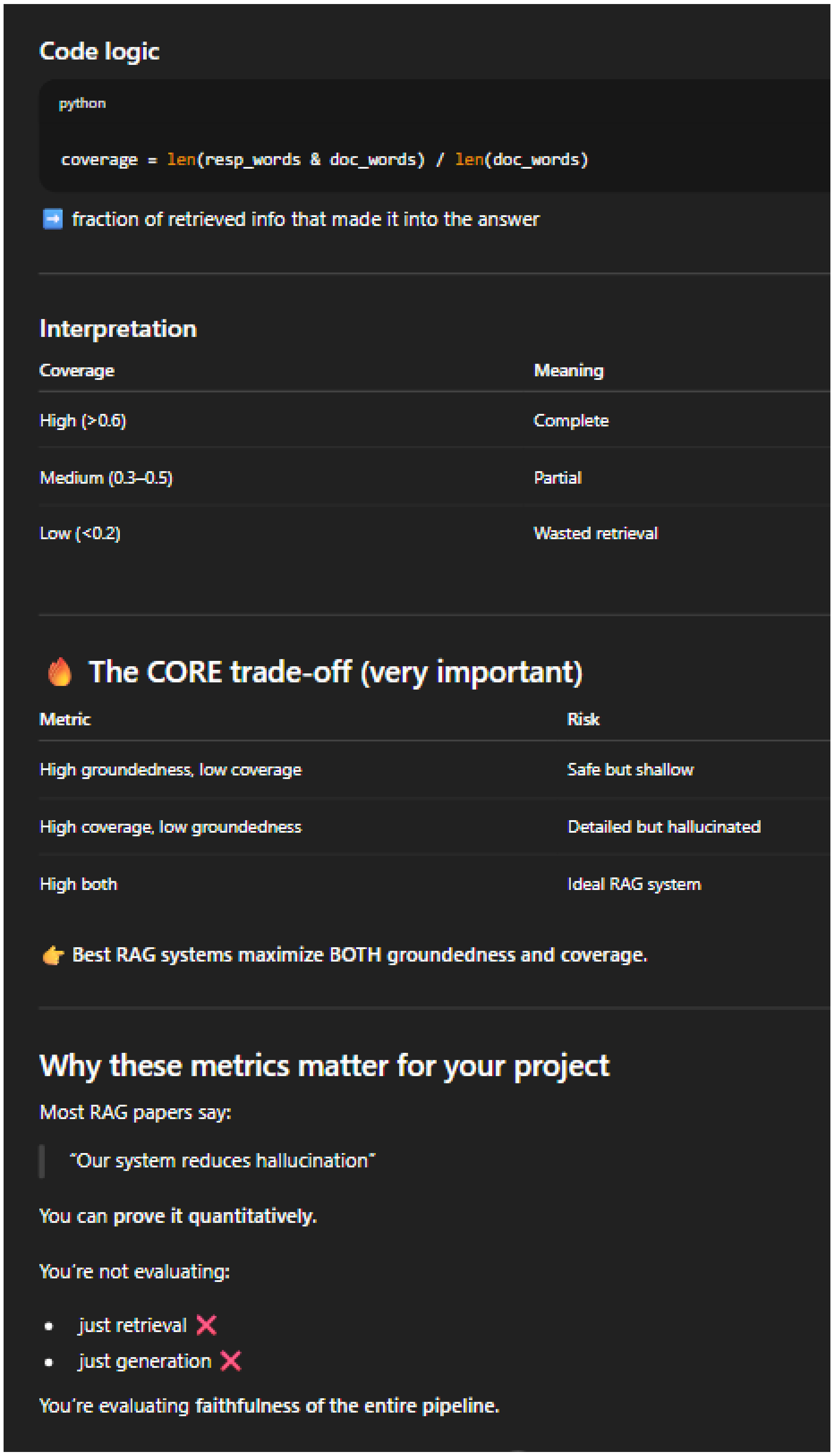
lua

```lua
{transformers, self-attention}
```

Coverage:

```
2 / 5 = 0.4
```

Answer is safe but incomplete.



Code logic

```python
coverage = len(resp_words & doc_words) / len(doc_words)
```

➡️ fraction of retrieved info that made it into the answer

Interpretation

Coverage	Meaning
High (>0.6)	Complete
Medium (0.3–0.5)	Partial
Low (<0.2)	Wasted retrieval

🔥 The CORE trade-off (very important)

Metric	Risk
High groundedness, low coverage	Safe but shallow
High coverage, low groundedness	Detailed but hallucinated
High both	Ideal RAG system

👉 **Best RAG systems maximize BOTH groundedness and coverage.**

Why these metrics matter for your project

Most RAG papers say:

> "Our system reduces hallucination"

You can **prove it quantitatively.**

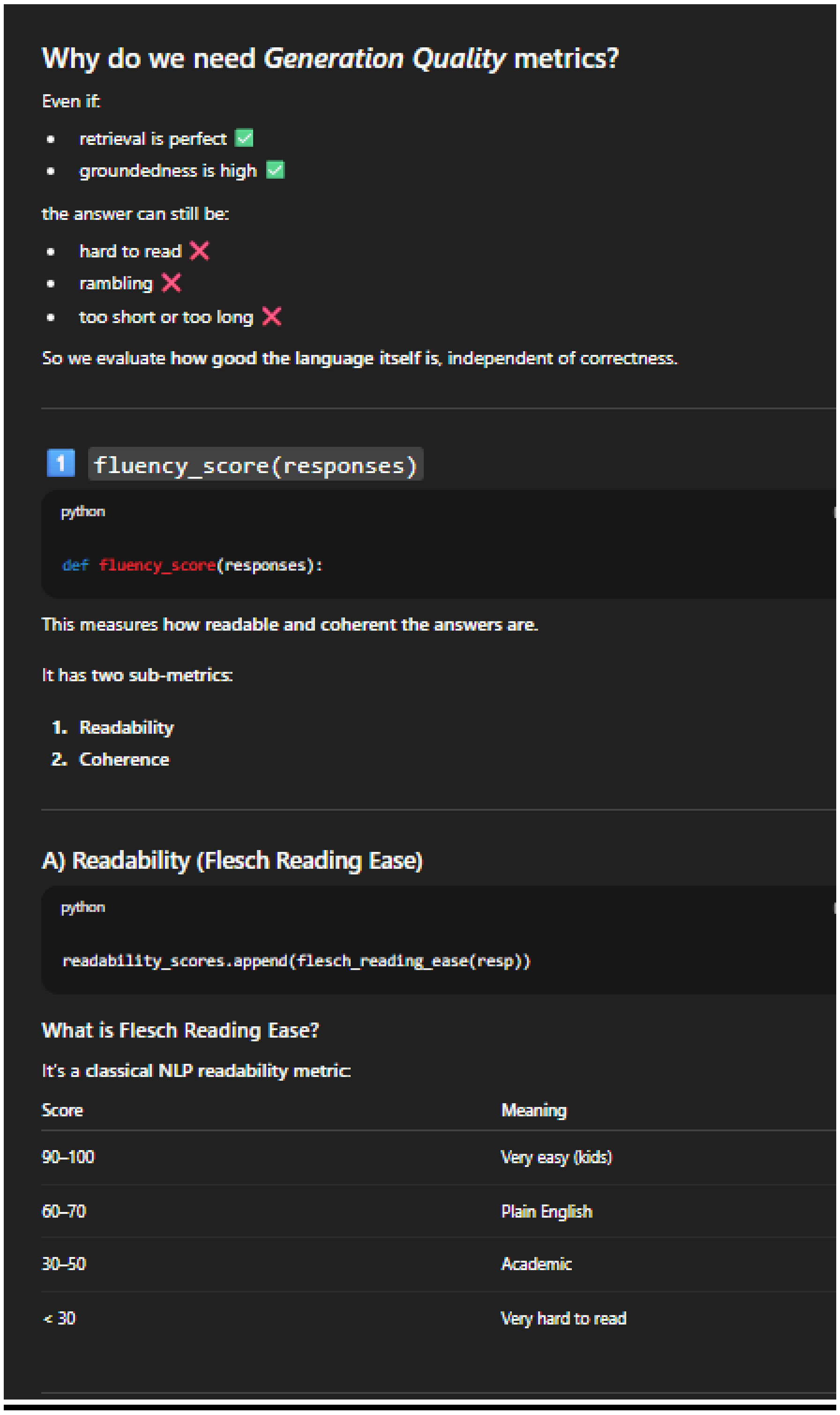
You're not evaluating:

- just retrieval ❌
- just generation ❌

You're evaluating **faithfulness of the entire pipeline.**

4. GENERATION QUALITY METRICS

```
def fluency_score(responses):
    """
    Readability and sentence structure quality
    """
    readability_scores = []
    coherence_scores = []

    for resp in responses:
        try:
            readability_scores.append(flesch_reading_ease(resp))
        except:
            readability_scores.append(0)

        sentences = nltk.sent_tokenize(resp)
        words = nltk.word_tokenize(resp)
        coherence = len(words) / len(sentences) if len(sentences) > 0 else
0
        coherence_scores.append(coherence)

    return {
        "Readability (Flesch)": np.mean(readability_scores),
        "Coherence (words/sentence)": np.mean(coherence_scores)
    }

def answer_length_stats(responses):
    """
    Simple length metrics
    """
    lengths = [len(resp.split()) for resp in responses]
    return {
        "Avg Answer Length": np.mean(lengths),
        "Min Length": np.min(lengths),
        "Max Length": np.max(lengths)
    }
```



Why do we need *Generation Quality* metrics?

Even if:

- retrieval is perfect ✅
- groundedness is high ✅

the answer can still be:

- hard to read ❌
- rambling ❌
- too short or too long ❌

So we evaluate how good the language itself is, independent of correctness.

1️⃣ `fluency_score(responses)`

```python
def fluency_score(responses):
```

This measures how readable and coherent the answers are.

It has two sub-metrics:

1. **Readability**
2. **Coherence**

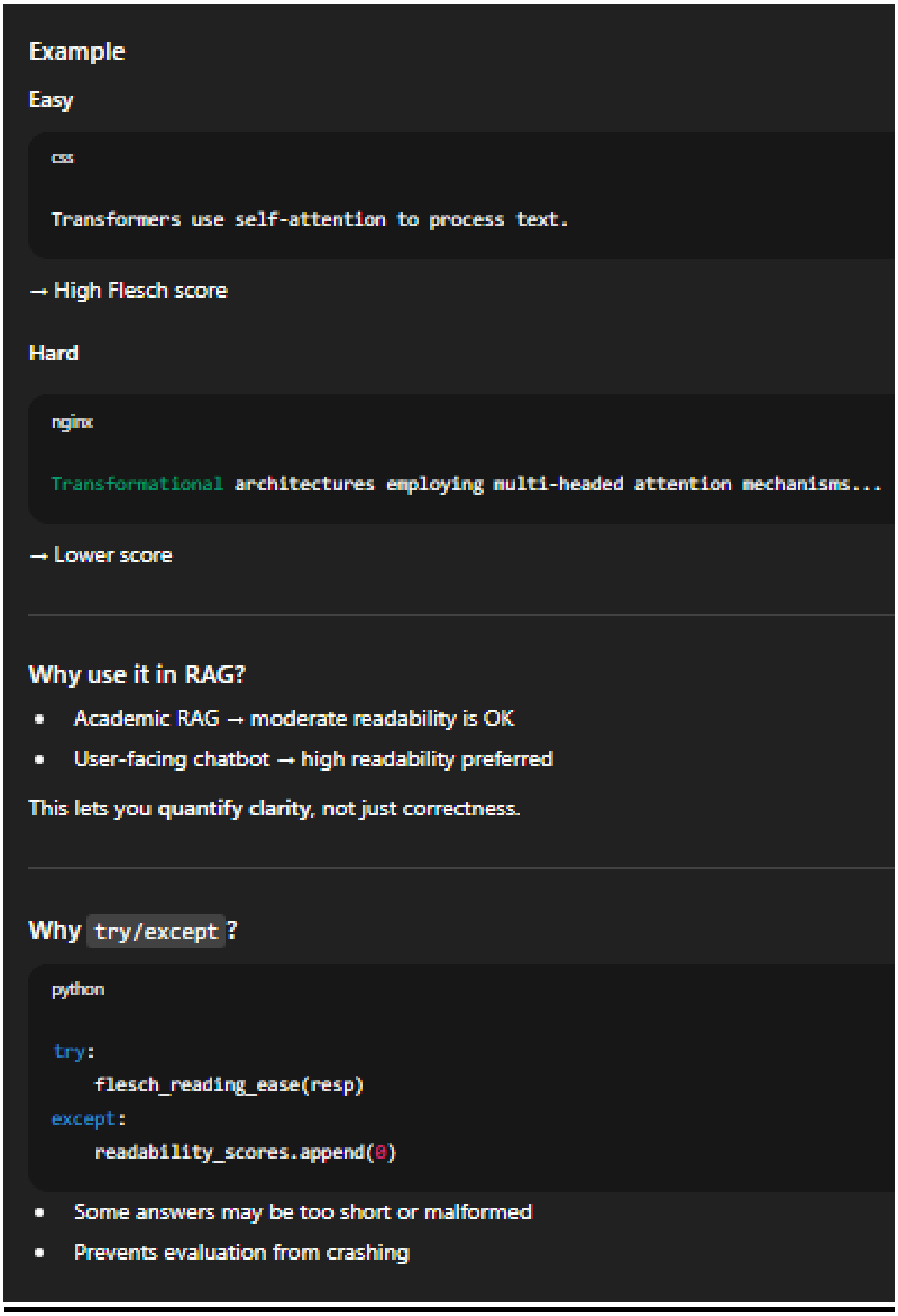
A) Readability (Flesch Reading Ease)

```python
readability_scores.append(flesch_reading_ease(resp))
```

What is Flesch Reading Ease?

It's a classical NLP readability metric:

Score	Meaning
90–100	Very easy (kids)
60–70	Plain English
30–50	Academic
< 30	Very hard to read



Example

Easy

```css
Transformers use self-attention to process text.
```

→ High Flesch score

Hard

```nginx
Transformational architectures employing multi-headed attention mechanisms...
```

→ Lower score

Why use it in RAG?

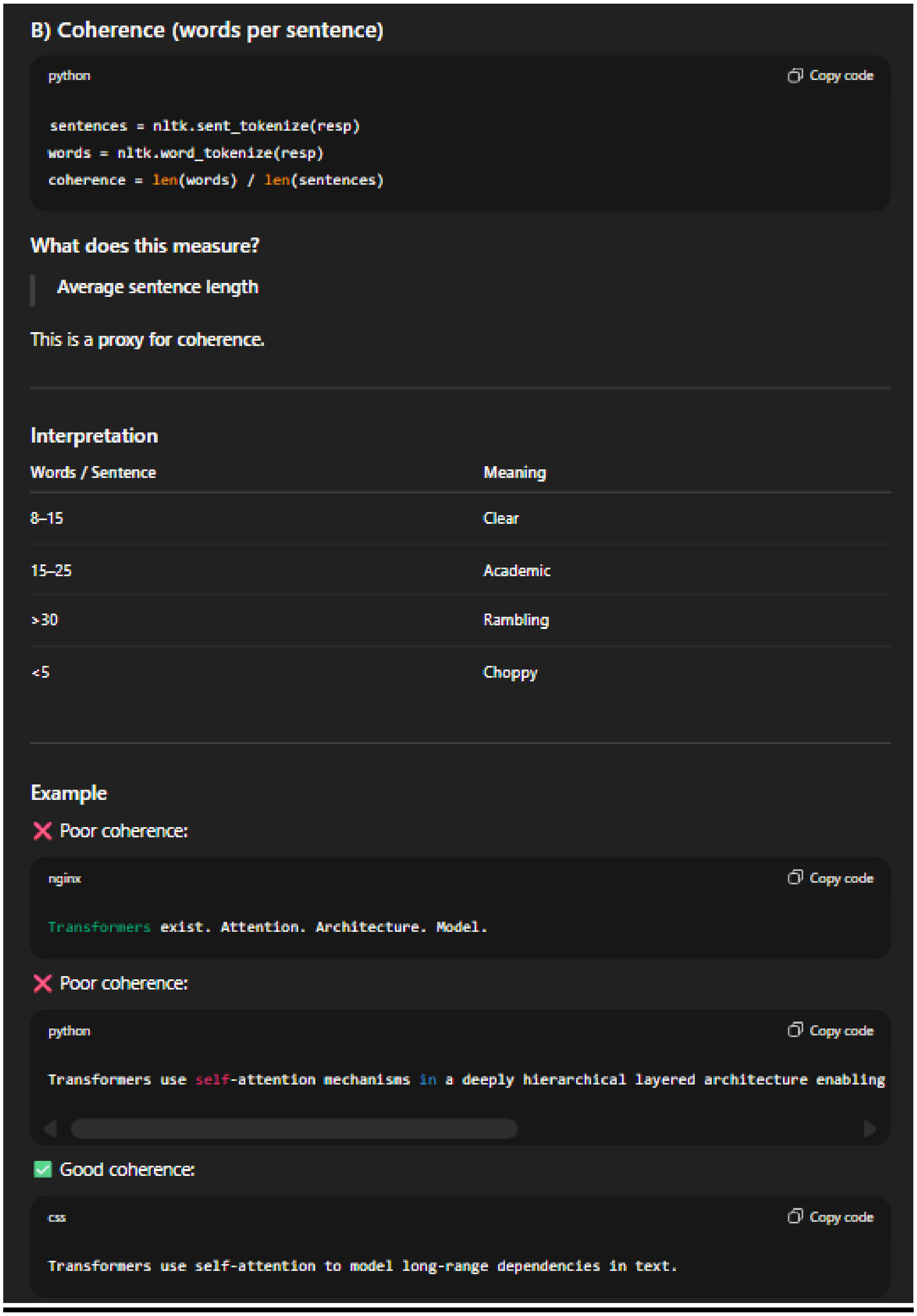
- Academic RAG → moderate readability is OK
- User-facing chatbot → high readability preferred

This lets you quantify clarity, not just correctness.

Why `try/except`?

```python
try:
    flesch_reading_ease(resp)
except:
    readability_scores.append(0)
```

- Some answers may be too short or malformed
- Prevents evaluation from crashing



B) Coherence (words per sentence)

```python
sentences = nltk.sent_tokenize(resp)
words = nltk.word_tokenize(resp)
coherence = len(words) / len(sentences)
```

What does this measure?

> Average sentence length

This is a proxy for coherence.

Interpretation

Words / Sentence	Meaning
8–15	Clear
15–25	Academic
>30	Rambling
<5	Choppy

Example

❌ Poor coherence:

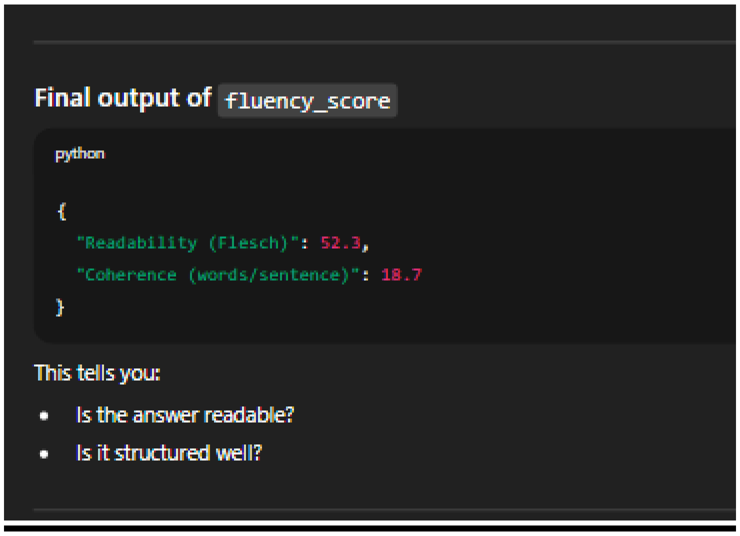
```nginx
Transformers exist. Attention. Architecture. Model.
```

❌ Poor coherence:

```python
Transformers use self-attention mechanisms in a deeply hierarchical layered architecture enabling
```

✅ Good coherence:

```css
Transformers use self-attention to model long-range dependencies in text.
```



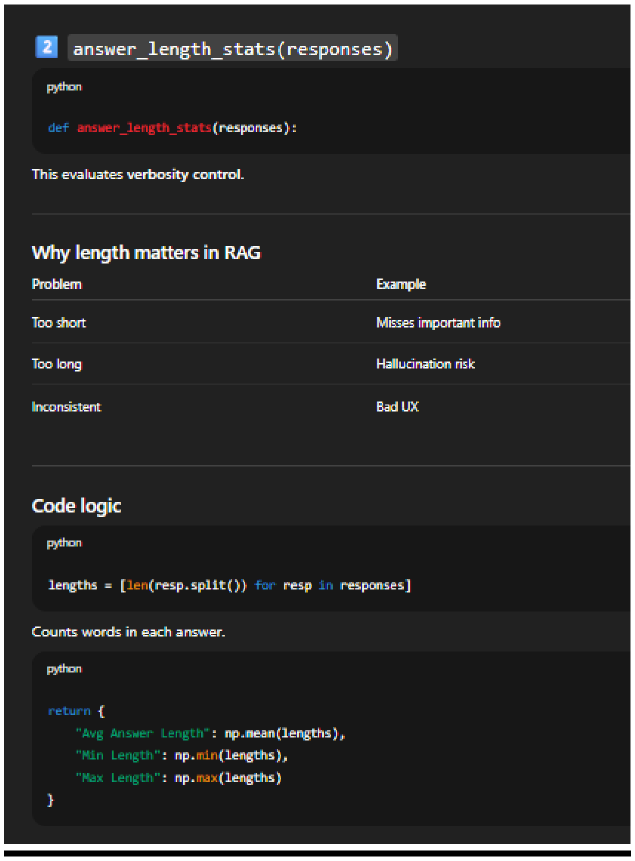
Final output of `fluency_score`

python

```python
{
  "Readability (Flesch)": 52.3,
  "Coherence (words/sentence)": 18.7
}
```

This tells you:

- Is the answer readable?
- Is it structured well?



2️⃣ `answer_length_stats(responses)`

python

```python
def answer_length_stats(responses):
```

This evaluates verbosity control.

Why length matters in RAG

Problem	Example
Too short	Misses important info
Too long	Hallucination risk
Inconsistent	Bad UX

Code logic

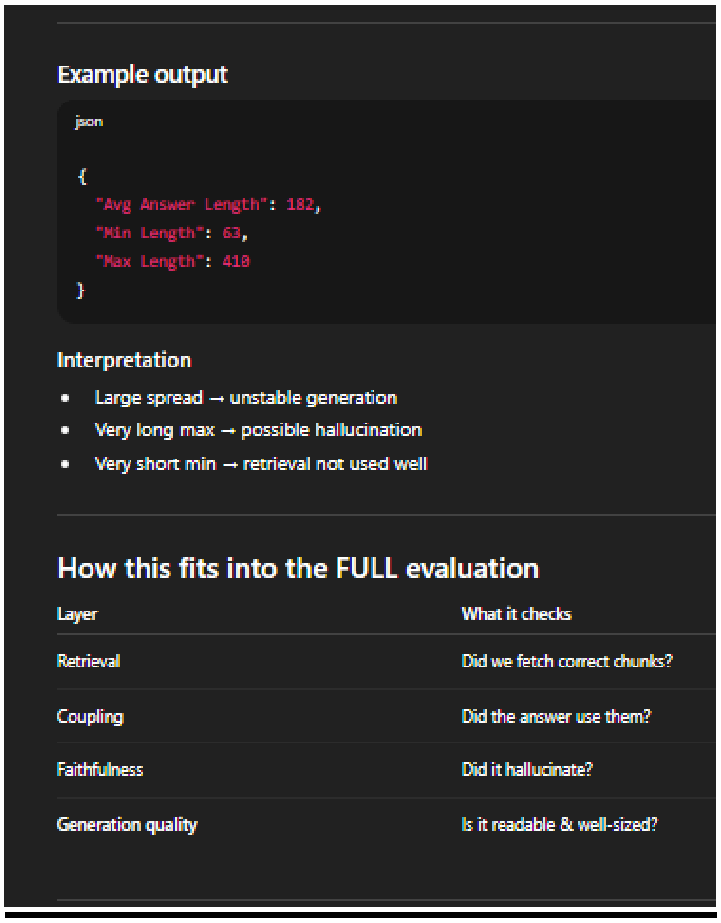
python

```python
lengths = [len(resp.split()) for resp in responses]
```

Counts words in each answer.

python

```python
return {
    "Avg Answer Length": np.mean(lengths),
    "Min Length": np.min(lengths),
    "Max Length": np.max(lengths)
}
```


Example output

```json
{
  "Avg Answer Length": 182,
  "Min Length": 63,
  "Max Length": 410
}
```

Interpretation

- Large spread → unstable generation
- Very long max → possible hallucination
- Very short min → retrieval not used well

How this fits into the FULL evaluation

Layer	What it checks
Retrieval	Did we fetch correct chunks?
Coupling	Did the answer use them?
Faithfulness	Did it hallucinate?
Generation quality	Is it readable & well-sized?